Université Mohammed V-Agdal Faculté des Sciences – Rabat

Département d'Informatique

Le module M_5 E_2 : SMIA

Algorithmique

Par

Pr. **Mohamed El Marraki**
marraki@fsr.ac.ma

2012/2013

Sommaire

1. Généralités sur l'Algorithmique

1.1 Introduction

L'**algorithmique** est un terme d'origine arabe, hommage à *Al Khawarizmi* (780-850) auteur d'un ouvrage décrivant des méthodes de calculs algébriques.
Un **algorithme** est une méthode de résolution de problème énoncée sous la forme d'une série d'opérations à effectuer. La mise en œuvre de l'algorithme consiste en l'écriture de ces opérations dans un langage de programmation et constitue alors la brique de base d'un programme informatique.

1 Une recette de cuisine est un algorithme!
2 Le mode d'emploi d'un magnétoscope est aussi un algorithme!
3 Indiqué un chemin à un touriste égaré ou faire chercher un objet à quelqu'un par téléphone c'est fabriquer - et faire exécuter - des algorithmes.

Un algorithme, c'est une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné.

1 Si l'algorithme est juste, le résultat est le résultat voulu, et le touriste se retrouve là où il voulait aller.
2 Si l'algorithme est faux, le résultat est, disons, aléatoire, et décidément, ce magnétoscope ne marche pas!

Pour fonctionner, un algorithme doit donc contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter (l'ordinateur).
L'ADN, qui est en quelque sorte le programme génétique, l'algorithme à la base de construction des êtres vivants, est une chaîne construite à partir de quatre éléments invariables. Ce n'est que le nombre de ces éléments, et l'ordre dans lequel ils sont arrangés, qui vont déterminer si on obtient une puce ou un éléphant.
Les ordinateurs eux-mêmes ne sont fondamentalement capables d'exécuter que quatre opérations logiques :

1 l'affectation de variables
2 la lecture / écriture
3 les tests
4 les boucles

Un algorithme informatique se ramène donc toujours au bout du compte à la combinaison de ces quatre petites briques de base. Il peut y en avoir quelques unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes.
La taille d'un algorithme ne conditionne pas en soi sa complexité : de longs algorithmes peuvent être finalement assez simples, et de petits algorithmes peuvent être très compliqués. L'informatique est la science du traitement automatique de l'information. Pour cela il faut:

1 modéliser cette information,
2 définir à l'aide d'un formalisme strict les traitements dont elle fera l'objet.
3 et enfin traduire ces traitements dans un langage compréhensible par un ordinateur.

Les deux premiers points concernent l'algorithmique, alors que le dernier point relève de ce que l'on nomme la ***programmation***.

L'écriture d'un programme consiste généralement à implanter une méthode de résolution déjà connue et souvent conçue indépendamment d'une machine pour fonctionner aussi bien sur toutes les machines ou presque. Ainsi, ce n'est pas le programme mais la méthode qu'il faut étudier pour comprendre comment traiter le problème. Le terme algorithme est employé en informatique pour décrire une méthode de résolution de problème programmable sur machine. Les algorithmes sont la « matière » de l'informatique et sont l'un des centres d'intérêt de la plupart, sinon la totalité, des domaines de cette science.

1.2 L'algorithmique

Principe

Définition : Un algorithme est une séquence bien définie d'opérations (calcul, manipulation de données, etc.) permettant d'accomplir une tache en un nombre fini de pas.

En principe un algorithme est indépendant de toute implantation. Cependant dans la pratique de la programmation il s'avère indispensable de tenir compte des capacités du langage de programmation utilisé.

La conception d'un algorithme passe par plusieurs étapes :

Analyse : définition du problème en terme de séquences d'opérations de calcul de stockage de données, etc. ;

Conception : définition précise des données, des traitements et de leur séquencement ;

Implantation : traduction et réalisation de l'algorithme dans un langage précis ;

Test : Vérification du bon fonctionnement de l'algorithme.

Remarque :

Les programmes sont souvent sur-optimisés. Il n'est pas toujours indispensable de se donner la peine de trouver l'implantation la plus efficace d'un algorithme, à mois que ce dernier ne soit susceptible d'être utilisé pour une tâche très répétitive. Dans les autres cas, une mise en œuvre simple conviendra souvent : on pourra être sûr que le programme fonctionnera, peut-être cinq ou dix fois moins vite que la version la plus optimisée, ce qui se traduira éventuellement par quelques secondes supplémentaires à l'exécution. En revanche, un mauvais choix d'algorithme peut entraîner une différence d'un facteur cent, mille ou plus, ce qui se traduira en minutes, en heures voir en jours au niveau des temps d'exécution.

Les caractéristiques d'un Algorithme

Un algorithme est une marche à suivre :

1. dont les opérations sont toutes définies et portent sur des objets appelés informations,
2. dont l'ordre d'exécution des opérations est défini sans ambiguïté,
3. qui est réputée résoudre de manière certaine un problème ou une classe de problèmes,
4. s'exprime dans un langage indépendant des langages de programmation,

1.3 L'algorithmique et la programmation

Un **programme** est la traduction d'un algorithme dans un certain langage de programmation. Il faut savoir qu'à chaque instruction d'un programme correspond une action du processeur.

1.3.1 Le but de la programmation :

- Utiliser l'ordinateur pour traiter des données afin d'obtenir des résultats.
- Abstraction par rapport au matériel (indépendance application / plate forme matérielle).
- Intermédiaire entre le langage machine (binaire) et le langage humain

1.3.2 Langages de programmation

Le langage utilisé par le processeur, est appelé **langage machine**. Il s'agit d'une suite de 0 et de 1 (du binaire). Toutefois le langage machine est difficilement compréhensible par l'humain. Ainsi il est plus pratique de trouver un langage intermédiaire, compréhensible par l'homme, qui sera ensuite transformé en langage machine pour être exploitable par le processeur. L'assembleur est le premier langage informatique qui ait été utilisé. Celui-ci est encore très proche du langage machine mais il permet déjà d'être plus compréhensible. Toutefois un tel langage est tellement proche du langage machine qu'il dépend étroitement du type de processeur utilisé (chaque type de processeur peut avoir son propre langage machine). Ainsi un programme développé pour une machine ne pourra pas être *porté* sur un autre type de machine (on désigne par le terme "**portable**" un programme qui peut être utilisé sur un grand nombre de machines). Pour pouvoir l'utiliser sur une autre machine il faudra alors parfois réécrire entièrement le programme!

Il y a trois catégories de langage de programmations : les **langages interprétés** et les **langages intermédiaires** et les **langages compilés**.

Langage interprété

Un langage de programmation est par définition différent du langage machine. Il faut donc le traduire pour le rendre intelligible du point de vue du processeur. Un programme écrit dans un langage interprété a besoin d'un programme auxiliaire (l'interpréteur) pour traduire au fur et à mesure les instructions du programme.
Exemples de langages interprétés : Le langage HTML (les pages web), le langage Maple (calcul mathématique), Prolog (Intelligence artificielle), etc.

Langage compilé :

Un programme écrit dans un langage dit "compilé" va être traduit une fois pour toutes par un programme annexe (le compilateur) afin de générer un nouveau fichier qui sera autonome, c'est-à-dire qui n'aura plus besoin d'un programme autre que lui pour s'exécuter (on dit d'ailleurs que ce fichier est exécutable).
Un programme écrit dans un langage compilé a comme avantage de ne plus avoir besoin, une fois compilé, de programme annexe pour s'exécuter. De plus, la traduction étant faite une fois pour toute, il est plus rapide à l'exécution.
Toutefois il est moins souple qu'un programme écrit avec un langage interprété car à chaque modification du fichier source il faudra recompiler le programme pour que les modifications prennent effet.
D'autre part, un programme compilé a pour avantage de garantir la sécurité du code source. En effet, un langage interprété, étant directement intelligible (lisible), permet à n'importe qui de connaître les secrets de fabrication d'un programme et donc de copier le code voire de le

modifier. Il y a donc risque de non-respect des droits d'auteur. D'autre part, certaines applications sécurisées nécessitent la confidentialité du code pour éviter le piratage (transaction bancaire, paiement en ligne, communications sécurisées, ...).
Exemples de langages compilés : Le langage C (Programmation système), le langage C++ (Programmation système objet), le Cobol (Gestion) etc.

Langages intermédiaires :

Certains langages appartiennent en quelque sorte aux deux catégories précédentes (LISP, Java, Python, ..) car le programme écrit avec ces langages peut dans certaines conditions subir une phase de compilation intermédiaire vers un fichier écrit dans un langage qui n'est pas intelligible (donc différent du fichier source) et non exécutable (nécessité d'un interpréteur). Les applets Java, petits programmes insérés parfois dans les pages Web, sont des fichiers qui sont compilés mais que l'on ne peut exécuter qu'à partir d'un navigateur Internet (ce sont des fichiers dont l'extension est .class).

Toutefois, à peu près tous les langages de programmation sont basés sur le même principe: Le programme est constitué d'une suite d'instructions que la machine doit exécuter. Celle-ci exécute les instructions au fur et à mesure qu'elle lit le fichier (donc de haut en bas) jusqu'à ce qu'elle rencontre une instruction (appelée parfois instruction de branchement) qui lui indique d'aller à un endroit précis du programme. Il s'agit donc d'une sorte de jeu de piste dans lequel la machine doit suivre le fil conducteur et exécuter les instructions qu'elle rencontre jusqu'à ce qu'elle arrive à la fin du programme et celui-ci s'arrête.

Historique des langages

- Langage de bas niveau (proche du langage machine):
  - Jusqu'en 1945 : langage binaire
  - 1950 : langage assembleur

- Langage de haut niveau (proche des langages naturels):
  Depuis 1955:
  - Programmation procédurale : Fortran, Cobol, Basic, Pascal, C, Ada…
  - Programmation orienté objet : SmallTalk, C++, Delphi, Java…
  - Programmation logique : Prolog…
  - Et beaucoup d'autres…

- Evolution:
  - Programmation impérative (fonction):
    - Exemples : Pascal, C, …
  - Programmation orientée objet (POO) :
    - Exemples : SmallTalk, Java, C++, …

1.3.3 La notion de fichier

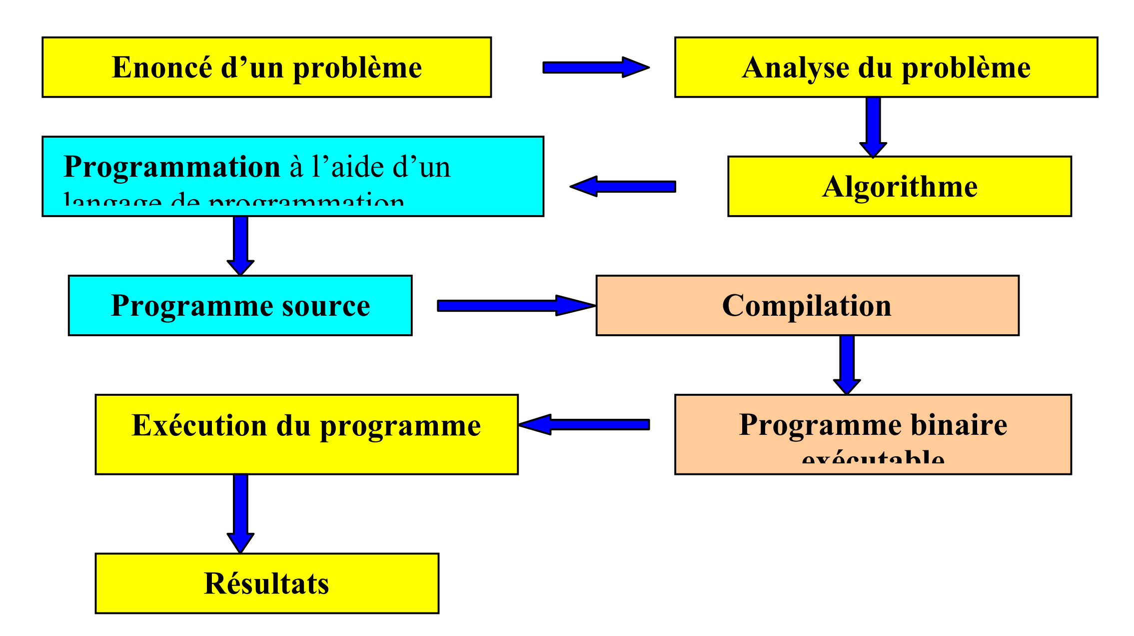
Dans un programme les instructions et données résident en mémoire centrale pour être exécutées, Les programmes et les données sont sauvegardées dans des fichiers qui portent des extensions spécifiques du langage :

- Les données et les programmes sont stockés dans des fichiers
- Un fichier est identifié par un nom et une extension (fichier.doc, pgcd.c, texte.tex, etc.)
- Un fichier est caractérisé par des attributs:
  - taille, date de création, date de modification, etc.…
- L'exploitation d'un fichier par une application se fait par l'intermédiaire du système d'exploitation qui accomplit les opérations logiques de base suivantes:
  - ouvrir, fermer un fichier
  - lire, écrire dans un fichier
- L'utilisateur peut créer, détruire, organiser, lire, écrire, modifier et copier des fichiers ou des enregistrements qui les composent

1.3.4 La démarche de programmation et analyse descendante

La résolution d'un problème passe par toute une suite d'étapes :

- Phase d'analyse et de réflexion (**algorithmique**)
- Phase de programmation
  - choisir un langage de programmation
  - traduction de l'algorithme en programme
  - programme (ou code) source
  - compilation : traduction du code source en code objet
  - traduction du code objet en code machine exécutable, compréhensible par l'ordinateur
- Phase de test
- Phase d'exécution



Le travail est ici surtout basé sur l'**analyse du problème** et l'écriture de l'**algorithme**.
La réalisation d'un programme passe par **l'analyse descendante** du problème : il faut réussir à trouver les actions élémentaires qui, en partant d'un environnement initial, nous conduisent à l'état final.

L'analyse descendante consiste à décomposer le problème donné en sous-problèmes, et ainsi de suite, jusqu'à descendre au niveau des primitives. Les étapes successives donnent lieu à des sous-algorithmes qui peuvent être considérés comme les primitives de machine intermédiaires (procédures en Pascal, fonction en C).

Le travail de l'analyse est terminé lorsqu'on a obtenu un algorithme ne comportant que :

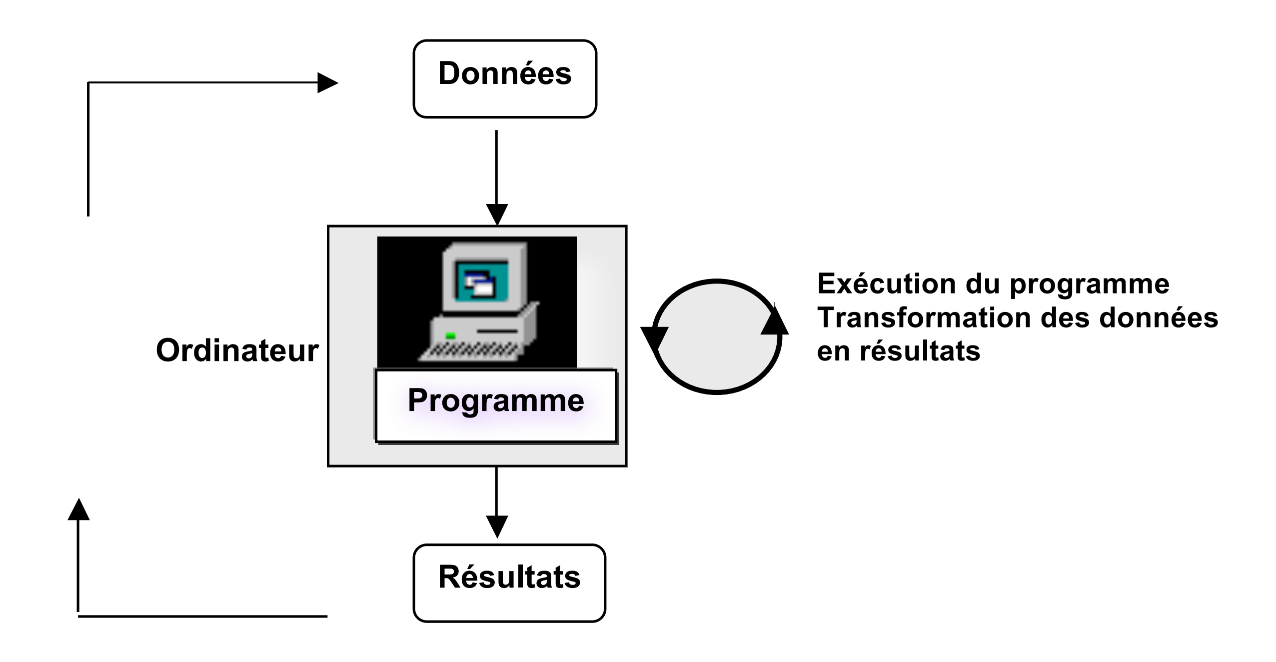
- Des primitives de la machine initiale,
- Des algorithmes déjà connus.

L'analyse descendante est la mise en pratique du *Discours de la méthode de Descartes*.
L'ordre des instructions est essentiel : la machine ne peut exécuter qu'une action à la fois et dans l'ordre donné; c'est la propriété de **séquentialité.**
Une fois ces actions déterminées, il suffit de les traduire dans le langage de programmation. Durant l'écriture d'un programme, on peut être confronté à 2 types d'erreur :

- o les erreurs syntaxiques : elles se remarquent à la compilation et sont le résultat d'une mauvaise écriture dans le langage de programmation.
- o les erreurs sémantiques : elles se remarquent à l'exécution et sont le résultat d'une mauvaise analyse. Ces erreurs sont beaucoup plus graves car elles peuvent se déclencher en cours d'exploitation du programme.

1.3.5 Exécuter un programme

La mise au point d'un programme informatique se fait en plusieurs étapes.



1.3.6 Pseudo langage

Un algorithme doit être lisible et compréhensible par plusieurs personnes. Il doit donc suivre des règles précises, il est composé d'une **entête** et d'un **corps** :

- l'entête, qui spécifie :
  - le nom de l'algorithme (**Nom :**)
  - son utilité (**Rôle :**)
  - les données "en entrée", c'est-à-dire les éléments qui sont indispensables à son bon fonctionnement (**Entrée :**)
  - les données "en sortie", c'est-à-dire les éléments calculés, produits, par l'algorithme (**Sortie :**)
  - les données locales à l'algorithmique qui lui sont indispensables (**Déclaration :**)
- le corps, qui est composé :
  - du mot clef **début**
  - d'une suite d'**instructions** indentées
  - du mot clef **fin**

Le plus important pour un algorithme sont les déclarations ainsi que les instructions qui constituent le corps de l'algorithme. Il existe des instructions qui ne servent qu'à la clarté de l'algorithme (l'ordinateur les ignore complètement), ce sont les **commentaires**.
Un commentaire a la syntaxe suivante :

```
/* ceci est un commentaire */
```

Exemple : voici le schéma d'un algorithme écrit en notre pseudo langage **:**

```
Nom        : le nom de l'algorithme                  }
Rôle       : que fait cet algorithme                 }
Entrée     : les données nécessaires                 }  Facultatifs
Sortie     : les résultats produits par l'algorithme }

Variables  : la déclaration des variables
Debut
      Instruction 1
      Instruction 2
      … … …          /* les commentaires explicatives des instructions */
      Instruction k
Fin
```

2. Variables

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier). Ces données peuvent être de plusieurs types : elles peuvent être des nombres, du texte, etc. Dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une **variable**.

2.1 Déclaration des variables

La première chose à faire avant de pouvoir utiliser une variable est de **créer la boîte et de lui coller une étiquette**. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites. C'est ce qu'on appelle la **déclaration des variables**.
Une variable ne peut être utilisée que s'elle est déclarée. La déclaration se fait par la donnée du **nom** de la variable et du **type** de la variable.

2.1.1 Noms de variables

Le nom de la variable (l'étiquette de la boîte) obéit à des règles qui changent selon le langage utiliser. Les principales règles à respecter sont :

- Le nom de variable peut comporter des lettres et des chiffres,
- On exclut la plupart des signes de ponctuation, en particulier les espaces.
- Un nom de variable doit commencer par une lettre.
- Le nombre maximal de caractères qui composent le nom d'une variable dépend du langage utilisé.
- Ne pas utiliser les mots clés du langage de programmation.

2.1.2 Types de variables

Lorsqu'on déclare une variable, il ne suffit pas de créer une boîte (réserver un emplacement mémoire) ; il faut préciser ce que l'on voudra mettre dedans, car de cela dépendent la **taille** de la boîte (l'emplacement mémoire) et le **type de codage** utilisé.

- **Types numériques classiques**

Commençons par le cas très fréquent, celui d'une variable destinée à recevoir **des nombres**.

- Si l'on réserve un octet pour coder un nombre, on ne pourra coder que $2^8 = 256$ valeurs différentes. Cela peut signifier par exemple les nombres entiers de 1 à 256, ou de 0 à 255, ou de –127 à +128.

- Si l'on réserve deux octets, on a droit à 2^{16} =65 536 valeurs ; avec trois octets, 2^{24} =16 777 216, etc.

Type Numérique	Plage
Octet	de 0 à 255
Entier simple	de -32768 à 32767
Entier double	de -2147483648 à 2147483647
Réel simple	de $-3.40x10^{38}$ à $-1.40x10^{-45}$ pour les négatives de $1,40x10^{-45}$ à $3.40x10^{38}$ pour les positives
Réel double	de $-1.79x10^{308}$ à $-4.94x10^{-324}$ les négatives de $4.94x10^{-324}$ à $1.79x10^{308}$ les positives

La syntaxe d'une déclaration de variable numérique en pseudo-langage aura la forme :

```
Variable g : Numérique

Variables PrixHT, TauxTVA, PrixTTC : Numérique
```

- **Type alphanumérique**

On dispose donc également du type alphanumérique (également appelé type caractère, type chaîne ou en anglais, le type string). Dans une variable de ce type, on stocke des caractères, qu'il s'agisse de lettres, de signes de ponctuation, d'espaces, ou même de chiffres. Le nombre maximal de caractères pouvant être stockés dans une seule variable string dépend du langage utilisé.

- Un groupe de caractères est appelé **chaîne** de caractères.
- En pseudo-code, une chaîne de caractères est toujours notée entre guillemets **" "**, car, il peut y avoir une confusion entre des nombres et des suites de chiffres. Par exemple, 423 peut représenter :
  - le nombre **423** (quatre cent vingt-trois),
  - ou la suite de caractères 4, 2, et 3 notée : **"423"**

La syntaxe d'une déclaration de variable de type alphanumérique en pseudo-langage aura la forme :

```
Variable nom : chaîne

Variables x, y : caractère.
```

- **Type booléen**

Le dernier type de variables est le type booléen : on y stocke uniquement les valeurs logiques VRAI et FAUX. On peut représenter ces notions abstraites de VRAI et de FAUX par tout ce qu'on veut : de l'anglais (TRUE et FALSE) ou des nombres (0 et 1). Le type booléen est très économique en termes de place mémoire occupée, un seul bit suffit.

En général dans un algorithme on trouve des déclarations de variables de la forme :

```
Variables a,b,c,delta,x,y : nombres
          nom,prenom : chaines de caractères
          ok : booleen.
```

3. Primitives

3.1 Affectation, expression et opérateurs

3.1.1 Affectation

Définition et notation :
L'affectation est l'action élémentaire dont l'effet est de donner une valeur à une variable (ranger une valeur à une place).
L'affectation est réalisée au moyen de l'opérateur ← (ou = en C et := en Pascal). Elle signifie " prendre la valeur se trouvant du côté droit (souvent appelée *rvalue*) et la copier du côté gauche (souvent appelée *lvalue*) ". Une rvalue représente toute constante, variable ou expression capable de produire une valeur, mais une lvalue doit être une variable distincte et nommée (autrement dit, il existe un emplacement physique pour ranger le résultat). Par exemple, on peut affecter une valeur constante à une variable (A ← 4), mais on ne peut pas affecter quoi que ce soit à une valeur constante - elle ne peut pas être une lvalue (on ne peut pas écrire 4 ← A).

Exemple :

X←3

Signifie mettre la valeur 3 dans la case identifiée par X. A l'exécution de cette instruction, la valeur 3 est rangée en X (nom de la variable).
La valeur correspond au contenu : 3
La variable correspond au contenant : X

On peut représenter la variable X par une boite ou case, et quand elle prend la valeur 3, la valeur 3 *est dans la case X* :

X [ ]	X [3]

On remarque qu'une variable ne peut contenir à un instant donné qu'une seule valeur.

Utilisations :
Voici quelques effets déclenchées par l'utilisation de l'affectation (←) :

Instructions	**actions**		**effets**	
X←3	X [ ]	3	X [3]	
X←2	X [3]	2	X [2]	plus de 3 !
Y←X	Y [ ]	X [2]	Y [2]	X [2]

La dernière instruction (Y← X) signifie : *copier dans Y la valeur actuelle de X.*

Un petit exercice instructif :
Quelles sont les valeurs successives prises par les variables X et Y suit aux instructions suivantes :
`X ← 1 ;  Y ← -4 ;  X ← X+3 ;  X ← Y-5 ;  Y ← X+2 ; Y ← Y-6 ;`

Réponses :

X	1	1	4	-9	-9	-9
Y		-4	-4	-4	-7	-13

Remarque :
À noter aussi que l'affectation est une expression comme une autre, c'est-à-dire qu'elle retourne une valeur. Il est donc possible d'écrire:

`X ← Y ← Z+2 ;`

ceci revenant à affecter à Y le résultat de l'évaluation de Z+2, puis à X le résultat de l'affectation Y ← Z+2, c'est-à-dire la valeur qu'on a donnée à Y. Remarquez l'ordre d'évaluation de la droite vers la gauche.

L'affectation des types primitifs est très simple. Puisque les données de type primitif contiennent une valeur réelle et non une référence à un objet, en affectant une valeur à une variable de type primitif on copie le contenu d'un endroit à un autre. Par exemple, si on écrit A ← B pour des types primitifs, alors le contenu de B est copié dans A. Si alors on modifie A, bien entendu B n'est pas affecté par cette modification. C'est ce qu'on rencontre généralement en programmation.

Echanger deux valeurs :
Problème : soit 2 variables quelconques (nombres ou caractères) x et y ayant respectivement comme valeur a et b ; quelles sont les affectations qui donneront à x la valeur b et à y la valeur a ?

Analyse : la première idée est d'écrire : x ← y ; y ← x. Mais ça ne marche pas, les deux variables se retrouvent avec la même valeur b ! Il faut mettre la valeur de x de coté pour ne pas la perdre : on utilise une variable auxiliaire z et on écrit les instructions suivantes :

`z ← x ;  x ← y ;  y ← z ;`

Le programme complet avec notre pseudo-langage est :

```
Nom        : échange
Rôle       : échanger deux valeurs
Entrée     : x et y
Sortie     : x et y
Variables x, y, z : quelconques
Debut
     x ← 3    /* initialisation de x et y */
     y ← -6
     z ← x    /* on stocke la valeur de x dans z */
     x ← y    /* on peut maintenant écrire dans x */
     y ← z    /* on remet l'ancien contenu de x dans y */
Fin
```

Vérification : il s'agit de vérifier que l'algorithme donne bien la solution voulu. Ecrivant après chaque instruction les valeurs des variables X, Y et Z :

```
x ← 3          " x = 3,   y =  ,    z =   "
y ← -6         " x = 3,   y = -6,   z =   "
z ← x          " x = 3,   y = -6,   z = 3 "
x ← y          " x = -6,  y = -6,   z = 3 "
y ← z          " x = -6,   y = 3,   z = 3 "  donc tout va bien.
```

Autre méthode : s'il s'agit de nombres entiers, nous pouvons nous passer d'une variable auxiliaire, mais en utilisant les primitives additionner et soustraire :

```
x ← a            " x = a,  y =   "
y ← b            " x = a,  y = b "
x  ←  x + y      " x = a + b,  y = b "
y  ←  x − y      " x = a + b,  y = a + b − b = a "
x  ←  x − y      " x = a + b − a = b, y = a "  donc tout va bien.
```

Le programme complet avec notre pseudo-langage est :

```
Nom        : échange_entiers
Rôle       : échanger deux valeurs entières
Entrée     : x et y
Sortie     : x et y
Variables x, y : nombres
Debut
     x ← 3      /* initialisation de x et y */
     y ← -6
     x ← x + y
     y ← x - y
     x ← x - y
Fin
```

3.1.2 Expression et opérateurs

Expression :
Dans une instruction d'affectation, on trouve :

- à gauche de la flèche, un nom de variable,
- à droite de la flèche, ce qu'on appelle une **expression** : *un ensemble de valeurs, reliées par des opérateurs, et équivalent à une seule valeur*
- L'expression située à droite de la flèche doit être du même type que la variable située à gauche.

Si l'un des trois points énumérés ci-dessus n'est pas respecté, la machine sera incapable d'exécuter l'affectation, et déclenchera une erreur.

Opérateurs :
Un opérateur est un signe qui relie deux valeurs, pour produire un résultat.

- **Opérateurs numériques** :

  Ce sont les quatre opérations arithmétiques :

  + addition
  - soustraction
  * multiplication
  / division

  Mentionnons également le ^ qui signifie "puissance". 45 au carré s'écrira donc 45^2.
  La multiplication et la division sont prioritaires sur l'addition et la soustraction.
    - 12*3+5 et (12*3)+5 valent strictement la même chose, à savoir 41.
    - En revanche, 12*(3+5) vaut 12*8 soit 96.

- **Opérateur alphanumérique : &**

  Cet opérateur permet de **concaténer** deux chaînes de caractères.

Exemple :

```
Nom        : concaténer
Rôle       : concaténer deux chaînes de caractères
Entrée     : A et B
Sortie     : C
Variables A, B, C : caractère
Début
     A ← "Bonjour"
     B ← " Tous le monde"
     C ← A & B
Fin
```

La valeur de C à la fin de l'algorithme est "Bonjour Tous le monde".

3.2 Lire et écrire

3.2.1 Introduction

Soit le programme suivant :

```
Variable A : entière
Début
   A ← 12^2
Fin
```

Ce programme nous donne le carré de 12 soit 144.

On remarque que :

- si l'on veut le carré d'un autre nombre que 12, il faut réécrire le programme.

- Le résultat est calculé par la machine elle le garde pour elle, et l'utilisateur qui exécute ce programme, ne saura jamais quel est le carré de 12.

C'est pourquoi, il faut utiliser des instructions qui permettent à l'utilisateur de dialoguer avec la machine.

3.2.2 Données et résultat

Pour pouvoir effectuer un calcul sur une variable, la machine doit connaître la valeur de cette variable. Si cette valeur n'a pas été déterminée par des initiations ou des calculs précédents, il faut que l'utilisateur lui fournisse, c'est une **donnée**. Il s'agit alors d'introduire une valeur à partir de l'extérieur de la machine et pour cela l'algorithme doit contenir l'instruction qui commande à la machine de **lire** la donnée.

Si un algorithme contenant l'instruction `X ← A^2` la machine ne peut exécuter cette instruction que si elle connaît la valeur de `A`, en supposant que la valeur de `A` en ce moment n'est pas connu, alors l'algorithme doit contenir l'instruction `lire(A)` qui signifie : mettre dans la case `A`, la valeur donnée par le **clavier** (organe d'entrée de la machine).

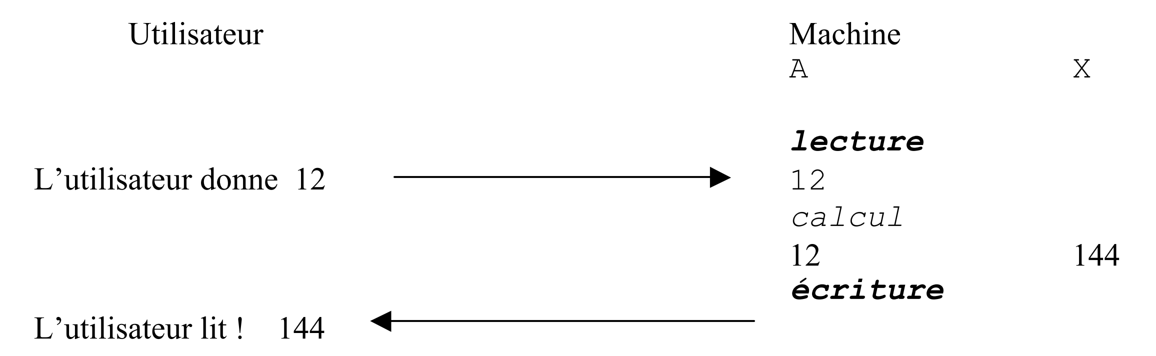
Dès que le programme rencontre une instruction **lire()**, l'exécution s'interrompt, attendant l'arriver d'une valeur par l'intermédiaire du clavier. Dès que la touche Entrée (Enter) a été frappée, l'exécution reprend.

Si on veut connaître le **résultat** d'un calcul ou le contenu d'une variable `X`, l'algorithme doit contenir l'instruction qui commande à la machine de fournir ce résultat. Cette instruction est **`écrire`**`(X)` qui signifie : mettre sur l'**écran** (organe de sortie de la machine) le contenu de la case `X`. Cette action ne modifie pas le contenu de `X`.

Exemple : soit le morceau d'algorithme suivant :
`A` étant une donnée, `X` un résultat

```
lire (A)
X ← A^2
écrire(X)
```

Schéma des actions effectuées par l'utilisateur et la machine :



La machine lit sur le clavier et écrit sur l'écran, l'utilisateur écrit sur le clavier et lit sur l'écran.

3.2.3 Les libellés

Avant de lire une variable, il est très fortement conseillé d'écrire des **libellés** à l'écran, afin de prévenir l'utilisateur de ce qu'il doit frapper :

```
écrire("Entrez votre nom : ")
lire(NomFamille)
```

3.2.4 Exemples

Exemple 1 :

Quel est le résultat produit par le programme suivant ?

```
Variables val, dval : entiers
Début
   val ← 234
   dval ← val * 2
   écrire(val)
   écrire(dval)
Fin
```

Réponses : 234 468

1ère amélioration :

```
Variables val, dval : entiers
Début
   écrire("donner un entier : ")  /* un libellé */
   lire(val)
   dval ← val * 2
   écrire("le double est : " )
   écrire(dval)
Fin
```

Exécution :

donner un entier : 234
le double est : 468

2ème amélioration :

```
Nom : double
Rôle : demande un nombre et affiche sont double
Entrée : val
Sortie : dval
Variables val, dval : entiers
Début
   écrire("donner un entier : ")
   lire(val)
   dval ← val * 2
   écrire(" le double de : ",val," est : ", dval )
Fin
```

Exécution :

donner un entier : 234
le double de : 234 est : 468

Exemple 2 :

Problème :

Multiplier deux nombres entiers.
En utilisant les primitives suivantes :

`lire()`, `écrire()`, affecter (←), multiplier (*).

Solution :

Algorithme :

```
Nom      : multiplication
Rôle     : demander deux nombres et afficher leur multiplication
Entrée   : A et B
Sortie   : C
variables A, B, C : entiers
Début
        écrire(" entrer la valeur de A : ")
        lire(A)
        écrire(" entrer la valeur de B : ")
        lire(B)
        C ← A * B
        écrire(" le produit de ",A," et ",B," est : ",C)
Fin
```

Exécution :

entrer la valeur de A : 12
entrer la valeur de B : -11
le produit de 12 est de 11 est : -132

3.2.5 Exercices :

Exercice 1 :

Quelles seront les valeurs des variables A, B et C après exécution des instructions suivantes ?

```
Variables A, B, C : Entier
  Début
     A ← 8
     B ← -2
     C ← A + B
     A ← 4
     C ← B – A
  Fin
```

Exercice 2 :

Quelles seront les valeurs des variables A et B après exécution des instructions suivantes ?

```
Variables A, B : Entier
  Début
     A ← 2
     B ← A + 5
     A ← A + B
     B ← B + 2
     A ← B – A
  Fin
```

Exercice 3 :

Que produit l'algorithme suivant ?

```
Variables A, B : Entier
Début
    écrire("entrer la valeur de A : ")   lire(A)
    écrire("entrer la valeur de B : ")   lire(B)
    A ← A + B
    B ← A - B
    A ← A – B
    écrire(" A = ",A) écrire(" B = ",B)
Fin
```

Exercice 4 :

Que produit l'algorithme suivant ?

```
Variables A, B, C : chaîne de caractères
Début
    A ← "423"
    B ← "12"
    C ← A & B
    écrire(" C = ",C)
Fin
```

Exercice 5 :

1. Ecrire un algorithme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable.
2. On dispose de trois variables A, B et C. Ecrivez un algorithme transférant à A la valeur de B, à B la valeur de C et à C la valeur de A (quels que soient les contenus préalables de ces variables).

Exercice 6 :

Ecrivez un algorithme qui calcule et affiche la surface et la circonférence d'un cercle (2πr et πr^2). L'algorithme demandera à l'utilisateur d'entrer la valeur du rayon.

Exercice 7 :

Comment calculer le plus rapidement possible x^{16} ?

Calculer x^{25} avec le minimum de multiplication.

Exercice 8 :

Ecrivez un algorithme qui calcule et affiche la surface et la circonférence d'un cercle (2πr et πr^2). L'algorithme demandera à l'utilisateur d'entrer la valeur du rayon.

Exercice 9 :

Écrire un algorithme qui effectue la lecture du temps `t` en seconde, et il affiche le temps `t` en jours, heure, minutes, secondes.

Exemple : si `t=21020` secondes l'algorithme affichera 0 jours 5 heures 50 minutes et 20 secondes.

3.5.4 si .. alors .., si .. alors .. sinon ..

Les primitives que nous allons présenter maintenant vont permettre à la machine de "choisir" les exécutions suivant les valeurs des données. Lors de l'exécution l'algorithme, la primitive :

```
si C alors
    A
finsi
```

Où C est une condition (on précisera plus loin la nature de cette condition) et A une instruction ou une suite d'instructions, a pour effet de faire exécuter A si et seulement si C est satisfaite.
La primitive

```
si C alors
    A
sinon B
finsi
```

A pour effet de faire exécuter A si C est satisfaite ou bien B dans la cas contraire (C non satisfaite).

Une **condition** est une comparaison. C'est-à-dire qu'elle est composée de trois éléments

- une valeur
- un opérateur de comparaison
- une autre valeur

Les **valeurs** peuvent être a priori de n'importe quel type (numériques, caractères…)
Les **opérateurs de comparaison** sont : = != < > =< >=
L'ensemble constitue donc si l'on veut une affirmation, qui a un moment donné est **VRAIE** ou **FAUSSE**.
A noter que ces opérateurs de comparaison s'emploient tout à fait avec des caractères. Ceux-ci sont codés par la machine dans l'ordre alphabétique, les majuscules étant systématiquement placées avant les minuscules. Ainsi on a :

```
"t" < "w"             VRAI
"Maman" > "Papa"      FAUX
"maman" > "Papa"      VRAI.
```

Conditions composées

Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme simple exposée ci-dessus. Prenons le cas " n est compris entre 5 et 8 ". En fait cette phrase cache non une, mais deux conditions. Car elle revient à dire que " n est supérieur à 5 et n est inférieur à 8 ". Il y a donc bien là deux conditions, reliées par ce qu'on appelle un opérateur logique, le mot **ET**.
Comme on l'a évoqué plus haut, l'informatique met à notre disposition trois opérateurs logiques : **ET**, **OU**, et **NON**.
Le ET a le même sens en informatique que dans le langage courant. Pour que C_1 ET C_2 soit VRAI, il faut impérativement que C_1 soit VRAIE et que C_2 soit VRAIE.

Il faut se méfier un peu plus du OU. Pour que C_1 OU C_2 soit VRAI, il suffit que C_1 soit VRAIE ou que C_2 soit VRAIE.
Le point important est que si C_1 est VRAIE et C_2 est VRAIE, alors C_1 OU C_2 est VRAIE. Le OU informatique ne veut donc pas dire " ou bien ".
VRAI ⇔ NON FAUX
On représente tout ceci dans des tables de vérité :

ET	V	F
V	V	F
F	F	F

OU	V	F
V	V	V
F	V	F

Exemple :

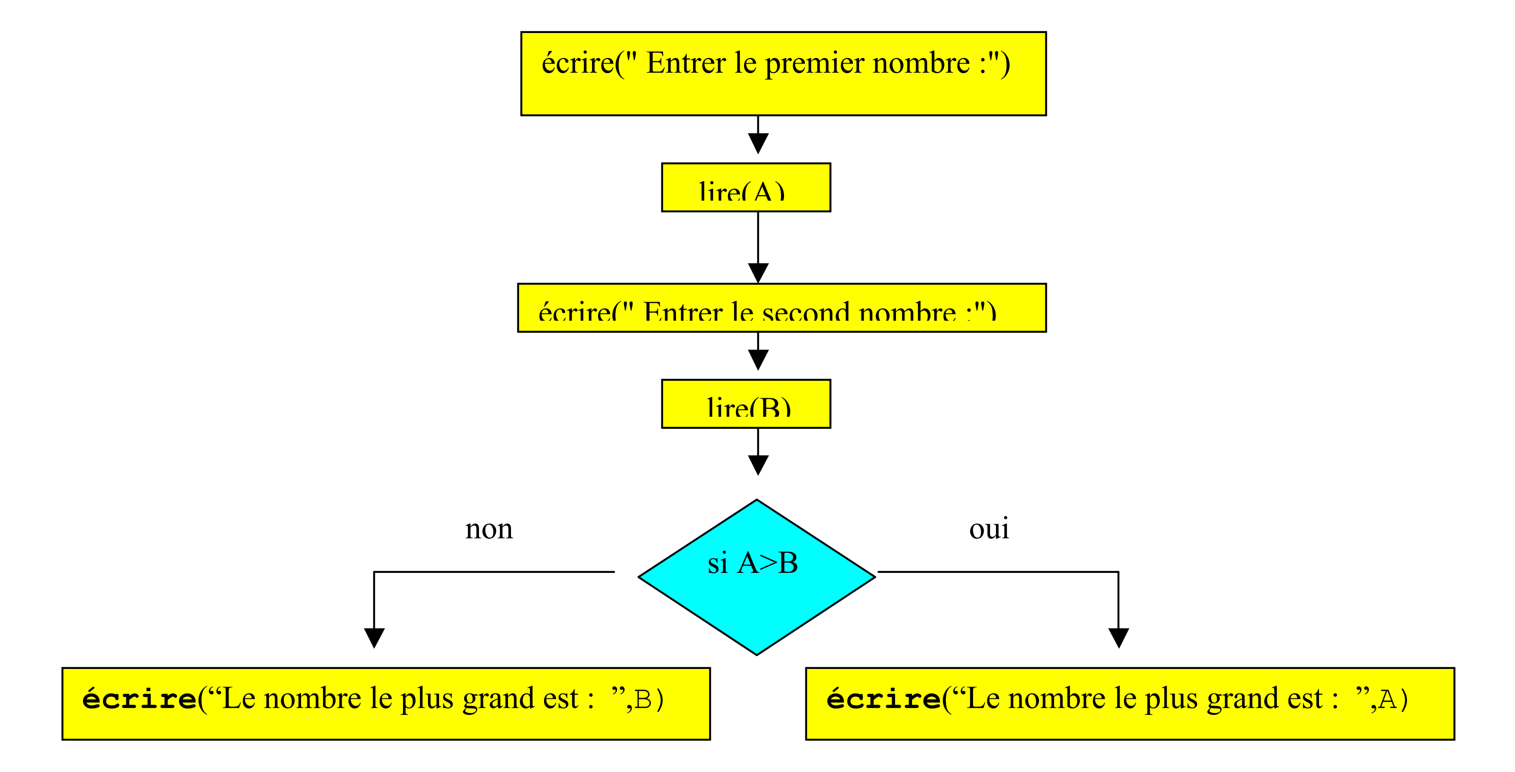
Problème :
Étant donnés deux nombres entiers positifs, identifier le plus grand des deux nombres.

Solution :

Analyse : si A>B alors le plus grand est A sinon le plus grand est B.
Conception : Algorithme

```
variables A, B : entiers
début
      écrire ("Programme permettant de déterminer le plus grand de deux entiers positifs”)
      écrire(“Entrer le premier nombre : ”)
      lire(A)
      écrire (“Entrer le second nombre : ”)
      lire(B)
      si  (A>B) alors
            écrire(“Le nombre le plus grand est :  ”,A)
      sinon
            écrire(“Le nombre le plus grand est : ”, B)
      finsi
fin
```

Organigramme



Tests imbriqués

Graphiquement, on peut très facilement représenter un **si** comme un aiguillage de chemin de fer. Un **si** ouvre donc deux voies, correspondant à deux traitements différents. Mais il y a des tas de situations où deux voies ne suffisent pas. Par exemple, un programme devant donner l'état de l'eau selon sa température doit pouvoir choisir entre trois réponses possibles (solide, liquide ou gazeuse).

Exemple :

```
Variable Temp : entier
Début
   écrire("Entrez la température de l'eau :")
   lire(Temp)
   si Temp =< 0 Alors
      écrire("C'est de la glace")
   finsi
   si Temp > 0 Et Temp < 100 Alors
     écrire("C'est du liquide")
   finsi
   si Temp > 100 Alors
      écrire("C'est de la vapeur")
   finsi
Fin
```

Les tests successifs portent sur une la même chose, la température (la valeur de la variable Temp). Il serait ainsi bien plus rationnel d'**imbriquer** les tests de cette manière :

Exemple :

```
Variable Temp en Entier
Début
   écrire("Entrez la température de l'eau :" )
   lire(Temp)
   si Temp =< 0 Alors
      écrire("C'est de la glace")
   sinon
      si Temp < 100 Alors
         écrire("C'est du liquide")
      sinon
         écrire("C'est de la vapeur")
       finsi
   finsi
Fin
```

Nous avons fait des économies au niveau de la frappe du programme : au lieu de devoir taper trois conditions, dont une composée, nous n'avons plus que deux conditions simples. Mais aussi, et surtout, nous avons fait des économies sur le temps d'exécution de l'ordinateur. Si la température est inférieure à zéro, celui-ci écrit dorénavant " C'est de la glace " et passe **directement** à la fin, sans être ralenti par l'examen d'autres possibilités (qui sont forcément fausses).
Cette deuxième version n'est donc pas seulement plus simple à écrire et plus lisible, elle est également plus performante à l'exécution.
Les structures de tests imbriqués sont donc un outil indispensable à la simplification et à l'optimisation des algorithmes.

Exercices :

Exercice 10 :

Ecrivez un algorithme qui donne le maximum de trois nombres saisis au clavier. Effectuez des tests pour :
2 5 8
3 1 3
8 -6 1

Exercice 11 :

Ecrivez un algorithme qui demande deux nombres à l'utilisateur et l'informe ensuite si leur produit est négatif, positif ou nul (**attention :** on ne doit pas calculer le produit des deux nombres).

Exercice 12 :

Écrivez un algorithme qui permet de discerner une mention à un étudiant selon la moyenne de ses notes :
- "Très bien" pour une moyenne comprise entre 16 et 20 (16<= moyenne <=20)
- "Bien" pour une moyenne comprise entre 14 et 16 (14<= moyenne <16)
- "Assez bien" pour une moyenne comprise entre 12 et 14 (12<= moyenne <14)
- "Passable" pour une moyenne comprise entre 10 et 12 (10<= moyenne <12)

Exercice 13 :
Écrivez un algorithme qui permet de résoudre une équation du second degré
$(a x^2 + b x + c = 0 \text{ avec } a \neq 0)$

Exercice 14 :
Les étudiants ayant passé l'examen d'algorithmique en session de Juin ont été classés selon leurs notes en trois catégories :

- pour une note inférieure strictement à 5, l'étudiant est éliminé,
- pour une note supérieure ou égale à 5 et inférieur strictement à 10, l'étudiant passe la session de rattrapage,
- pour une note supérieure ou égale à 10, l'étudiant valide le module

Ecrivez un algorithme qui demande à l'utilisateur d'entrer la note du module, puis affiche la situation de l'étudiant selon sa note (on suppose que l'utilisateur entre une note valide entre 0 et 20).

3.5.5 Les Boucles

La notion d'itération (boucle) est une des notions fondamentales de l'algorithmique. On l'utilise souvent quand on doit exercer plusieurs fois le même traitement sur un même objet, ou plusieurs objets de même nature. Mais son réel intérêt réside dans le fait que l'on peut modifier, à chaque répétition, les objets sur lesquels s'exerce l'action répétée.

Pour comprendre l'intérêt des boucles, on se place dans un cas bien précis :
Prenons le cas d'une saisie au clavier (une lecture), par exemple, on pose une question à laquelle l'utilisateur doit répondre par O (Oui) ou N (Non). Mais l'utilisateur maladroit risque de taper autre chose que O ou N. Dès lors, le programme peut soit planter par une erreur d'exécution (parce que le type de réponse ne correspond pas au type de la variable attendu) soit se dérouler normalement jusqu'au bout, mais en produisant des résultats faux.
Alors, dans tout programme, on met en place ce qu'on appelle un **contrôle de saisie** (pour vérifier que les données entrées au clavier correspondent bien à celles attendues par l'algorithme).
On pourrait essayer avec un **si**. Voyons voir ce que ça donne :

```
Variable Rep Caractère
écrire("Voulez vous un café ? (O/N)")
lire(Rep)
si Rep != "O" et Rep != "N" alors
   écrire("Saisie erronnée. Recommencez")
   Lire(Rep)
finsi
```

Ça marche tant que l'utilisateur ne se tromper qu'une seule fois, et il rentre une valeur correcte à la deuxième demande. Si l'on veut également éviter une deuxième erreur, il faudrait rajouter un SI. Et ainsi de suite, on peut rajouter des centaines de SI. Mais cela ne résout pas le problème. La seule issue est l'utilisation d'une **boucle**.

Il existe trois façons d'exprimer algorithmiquement l'itération :

- **TantQue**
- **Répéter … jusqu'à …**
- **Pour … jusqu'à …**

La boucle TantQue

Le schéma de la boucle TantQue est :

```
TantQue conditions
    ...
    Instructions
    ...
FinTantQue
```

Le principe est simple : le programme arrive sur la ligne du TantQue. Il examine alors la valeur de la condition. Si cette valeur est VRAI, le programme exécute les instructions qui suivent, jusqu'à ce qu'il rencontre la ligne FinTantQue. Il retourne ensuite sur la ligne du TantQue, procède au même examen, et ainsi de suite. On ne s'arrête que lorsque la condition prend la valeur FAUX.
Illustration avec notre problème de contrôle de saisie :

```
Variable Rep en Caractère
Ecrire "Voulez vous un café ? (O/N)"
TantQue Rep != "O" ET Rep != "N"
    Lire Rep
    Si Rep != "O" ET Rep != "N" Alors
        Ecrire "Saisie erronée. Recommencez"
    FinSi
FinTantQue
```

La boucle Répéter … jusqu'à …

Le schéma de la boucle répéter est :

```
Répéter
    ...
    Instructions
    ...
jusqu'à conditions
```

Le principe est simple : toutes les instructions écrites entre `Répéter` et `jusqu'à` sont exécutées au moins une fois et leur exécution est répétée jusqu'à ce que la condition placée derrière jusqu'à soit satisfaite.
Illustration avec notre problème de contrôle de saisie :

```
Variable Rep en Caractère
Ecrire "Voulez vous un café ? (O/N)"
Répéter
    Lire Rep
    Si Rep != "O" ET Rep != "N" Alors
        Ecrire "Saisie erronée. Recommencez"
   FinSi
Jusqu'à Rep = "O" OU Rep = "N"
```

La boucle Pour … jusqu'à …

Cette boucle est utile surtout quand on connaît le nombre d'itérations à effectuer.

Le schéma de la boucle Pour est :

> **Pour** i allant de début **jusqu'à** fin
> ...
> Instructions
> ...
> **FinPour**

Le principe est simple :

- on initialise i par début
- on test si on a pas dépassé fin
- on exécute les instructions
- on incrémente i (i← i + 1)
- on test si on a pas dépassé fin
- etc.

Exemple :

Problème :
On veut écrire un algorithme qui affiche le message "Bonjour à tous" 100 fois.

Résolution :

Au lieu d'écrire l'instruction :

```
écrire("Bonjour à tous") ;
```

100 fois. On utilise plutôt une boucle :

```
variable i entière
Pour i allant de 1 à 100 faire
     écrire("Bonjour à tous")
finpour
```

On peut améliorer ce programme par :

- ajouter un entier n : le nombre de fois que le message s'afficher à l'écran,
- afficher la variable i dans la boucle : pour numéroter les passages dans la boucle.

```
variable n, i entières
écrire("entrer le nombre n :")
```

```
lire(n)
Pour i allant de 1 à n faire
     écrire("Bonjour à tous la ",i," fois")
finpour
```

Dans la boucle précédente le i est incrémenté automatiquement. Si on désire utiliser la boucle TantQue, il faut incrémenter le i soit même :

```
variable n, i entières
écrire("entrer le nombre n :")
lire(n)
i ←1
TantQue(i<=n) faire
     écrire("Bonjour à tous la ",i," fois")
     i ← i + 1
FinTantQue
```

Des boucles imbriquées

De même qu'une structure SI … ALORS peut contenir d'autres structures SI … ALORS, une boucle peut contenir d'autres boucles.

```
Variables i, j entier
Pour i allant de 1 à 10
    écrire("Première boucle") ;
    Pour j allant de 1 à 6
        écrire("Deuxième boucle") ;
    Finpour ;
Finpour ;
```

Dans cet exemple, le programme écrira une fois "`Première boucle`" puis six fois de suite "`Deuxième boucle`", et ceci dix fois en tout. A la fin, il y aura donc eu 10 x 6 = 60 passages dans la deuxième boucle (celle du milieu).

Notez la différence marquante avec cette structure :

```
Variables i, j entier
Pour i allant de 1 à 10
    écrire("Première boucle") ;
Finpour ;
Pour j allant de 1 à 6
    écrire("Deuxième boucle") ;
Finpour ;
```

Ici, il y aura dix écritures consécutives de "`Première boucle`", puis six écritures consécutives de "`Deuxième boucle`" , et ce sera tout.

Examinons l'algorithme suivant :

```
Variable i  entier
Pour i allant de 1 à 10
      i ← i * 2
      écrire("Passage numéro : ",i) ;
Finpour
```

On remarque que la variable i est gérée "en double", ces deux gestions étant contradictoires. D'une part, la ligne " Pour … " augmente la valeur de i de 1 à chaque passage. D'autre part la ligne " i ← i * 2" double la valeur de i à chaque passage. Il va sans dire que de telles manipulations perturbent complètement le déroulement normal de la boucle.

Exemple :

Problème :

On veut écrire un algorithme qui calcul la somme des entiers positifs inférieurs ou égaux à N.

Résolution :

1ère étape : Analyse

1. Entrer la valeur de N
2. Calculer la somme des N premiers entiers positifs
3. Afficher le résultat

2ème étapes : Conceptions

1.

```
déclaration des variables N, i, somme : entiers
écrire(donner la valeur de N)
lire(N)
si N<0 alors erreur
initialiser somme et i
Répéter
     somme ← somme + i
     i← i+1
jusqu'à i>=N
écrire("la somme est ",somme)
```

2.

```
déclaration des variables N, i, somme : entiers
écrire(donner la valeur de N)
lire(N)
si N<0 alors erreur
initialiser somme et i
TantQue i<= N
     somme ← somme + i
     i← i+1
FinTantque
Ecrire("la somme est ",somme)
```

3.

```
déclaration des variables N, i, somme : entiers
```

```
écrire(donner la valeur de N)
lire(N)
si N<0 alors erreur
initialiser somme et i
Pour i allant de 1 à N
     somme ← somme + i
FinPour
Ecrire("la somme est ",somme)
```

3ème étape : Test

Somme_N_entiers
Donner N : -1
N doit etre >0 !
Somme_N_entiers
Donner N : 7845
La somme est : 30775935
Somme_N_entiers
Donner N : 10
La somme est : 55

Remarque : Pour cet exemple on peut faire une vérification plus complète en calculant une autre variable “somme1” = N(N+1)/2 , qui est la somme 1+2+3+ …+N, et la comparée à “somme” ensuite afficher le résultat de la comparaison.

Méthodologie pour l'écriture d'une boucle :

- repérer une action répétitive, donc une boucle
- choix entre boucle avec compteur ou sans
  Question ? Peut-on prévoir/déterminer le nombre d'itérations ?
  - si oui, boucle avec compteur : la boucle `pour …`
  - si non, boucle sans compteur
    Est ce que il faut commencer l'action avant de tester ou l'inverse ?
    - si tester d'abord, alors boucle TantQue
    - si action puis tester, alors `Répéter … jusqu'à`
- écrire l'action répétitive et l'instruction de boucle choisie
  Question ? Faut-il préparer les données à l'itération suivante ?
    - si oui, compléter le corps de boucle
- initialiser les variables utilisées (si nécessaires)
- écrire les conditions d'arrêt, voire l'incrémentation de la variable de contrôle.
- exécuter pour les cas extrêmes et au moins un cas "normal".

3.6 Exemple

Ecrire l’algorithme qui compte le nombre de bits nécessaires pour coder en binaire un entier n.

- Le nombre de bits nécessaire pour coder l’entier n est [lg(n)] (l’entier juste au dessus du

logarithme à base 2 de l'entier n).
- **Analyse** : on initialise une variable `nb` à 0 et à chaque fois que l'on divise n par 2 on augment de 1 la valeur de nb, on répète ce procédé jusqu'à ce que le quotient obtenu est nul.
- **L'algorithme** :

```
Variables i,n,nb : entiers
Debut
    Ecrire(" Entrer la valeur de n :")
    lire(n)
    i ← n
    nb ← 0
    TantQue(i<>0) faire
          i ← i/2
          nb ← nb+1
    FinTantQue
   Ecrire("Pour coder ",n," en binaire il faut ",nb,"bits")
Fin
```

- **Exécution** :

Entrer la valeur de n : 13
Pour coder 13 en binaire il faut 4 bits
=============================
Entrer la valeur de n : 1750
Pour coder 1750 en binaire il faut 11 bits
==============================
Entrer la valeur de n : 0
Pour coder 0 en binaire il faut 0 bits
Erreur !!!!!

- **Amélioration :**

```
Variables i,n,nb : entiers
 Debut
    Ecrire(" Entrer la valeur de n :")
    lire(n)
    i ← n/2   /* Pour le cas de zéro */
    nb ←  1
    TantQue(i<>0) faire
           i ← i/2
           nb ← nb+1
    FinTantQue
    Ecrire("Pour coder ",n," en binaire il faut ",nb,"bits")
 Fin
```

- **Implémentation en langage C** :

```
#include<stdio.h>
void main(){
      int i,n,nb ;
      printf(" Entrer la valeur de n :") ;
      scanf("%d",&n) ;
      i = n ;
      nb =  0 ;
      while(i!=0){
            i = i/2 ;
```

```
            nb = nb + 1;
        }
        printf("Pour coder %d en binaire il faut %d bits\n",n,nb) ;
    }
```

3.6 Exercices

Exercice 15 :

1. Écrivez un algorithme qui affiche 100 fois la phrase : "je ne dois pas arriver en retard en classe".
2. Écrivez un algorithme qui affiche les entiers de 1 à 100.
3. Écrivez un algorithme qui affiche les entiers pairs de 1 à 100.

Exercice 16 :

1. Écrivez un algorithme qui calcule la somme des n premiers nombres entiers positifs. L'algorithme demandera à l'utilisateur d'entrer la valeur de n.
2. Écrivez un algorithme qui calcule la somme des n premiers nombres entiers positifs paires. L'algorithme demandera à l'utilisateur d'entrer la valeur de n.

Exercice 17 :

1. Exécuter le programme suivant :

```
Variable i, j : Entier
debut
   Pour i←1 jusqu'à 5
      Ecrire(" i= ", i)
      Pour j←1 jusqu'à 3
         Ecrire("le produit de",i," et ",j," est:",i*j)
      FinPour
   FinPour
Fin
```

2. Exécuter le programme suivant :

```
Variable i, j : Entier
debut
   Pour i←1 jusqu'à 5
      Ecrire(" i= ", i)
   FinPour
   Pour j←1 jusqu'à 3
      Ecrire("le produit de",i," et ",j," est:",i*j)
   FinPour
Fin
```

Exercice 18 :

1. Écrivez un algorithme qui calcule la somme S suivante :

   $$S = 1^2 + 2^2 + 3^2 + \ldots + (n-1)^2 + n^2.$$

   L'algorithme demandera à l'utilisateur d'entrer la valeur de n.

2. Écrivez un algorithme qui calcule le factoriel de n :

   $$n\ ! = 1\times2\times3\times \dots \times(n-1)\times n.$$

   L'algorithme demandera à l'utilisateur d'entrer la valeur de `n`.

Exercice 19 :

Soit l'algorithme suivant :

```
variables a,b,r : entiers
début
     écrire("donner les valeurs de a et b : ")
     lire(a,b)
     TantQue b>0 faire
          r ← a%b   /* a%b :reste de la division de a par b */
          a ← b
          b ← r
     FinTanQue
     écrire(a)
Fin
```

1. Exécuter l'algorithme (afficher dans un tableau les valeurs de a, b et r) pour :
   a. a = 50 et b = 45
   b. a = 21 et b = 13
   c. a = 96 et b = 81

2. Que fait l'algorithme précédant.

Exercice 20:

1. Un nombre entier p (différent de 1) est dit premier si ses seuls diviseurs positifs sont 1 et p. Ecrivez un algorithme qui effectue la lecture d'un entier `p` et détermine si cet entier est premier ou non.
2. Deux nombres entiers n et m sont qualifiés d'**amis**, si la somme des diviseurs de n est égale à m et la somme des diviseurs de m est égale à n (on ne compte pas comme diviseur le nombre lui même et 1).
   Exemple : les nombres 48 et 75 sont deux nombres **amis** puisque :
   Les diviseurs de 48 sont : 2, 3, 4, 6, 8, 12, 16, 24 et
   $$2 + 3 + 4 + 6 + 8 + 12 + 16 + 24 = 75$$
   Les diviseurs de 75 sont : 3, 5, 15, 25 et
   $$3 + 5 + 15 + 25 = 48.$$
   Ecrire un algorithme qui permet de déterminer si deux entiers n et m sont amis ou non.

4. Les structures de données statiques

4.1 Tableaux à une dimension

4.1.1 Introduction

Imaginons que dans un programme, nous ayons besoin simultanément de 25 valeurs (par exemple, des notes pour calculer une moyenne). La solution consiste à déclarer 25 variables réelles, appelées par exemple $n_1, n_2, n_3, \ldots, n_{25}$ et la variable `moyenne` réelle.

`moyenne = (`$n_1+n_2+n_3+\ldots+n_{25}$`) / 25`

En programmation (exemple langage C) : l'ordinateur va réserver `25*4=100` octets pour les valeurs réelles des 25 variables et `25*4=100` octets pour les adresses de ces 25 variables.
La programmation nous permet de rassembler toutes ces variables en une seule : " la note numéro 1 ", " la note numéro 2 ", …, " la note numéro 25 ".
Un ensemble de valeurs portant ainsi le même nom de variable et repérées par un nombre, s'appelle un **tableau**, et le nombre qui sert à repérer chaque valeur s'appelle un **indice**.
Un tableau de taille `n` est une structure très simple constituée de `n` emplacements consécutifs en mémoire. Il est donc possible d'accéder à un élément d'un tableau en temps constant pourvu que l'on connaisse sa position (ou indice). Un tableau est donc une structure très simple et très efficace. Il n'est cependant pas possible de modifier la taille d'un tableau, ce qui est gênant pour un certain nombre d'algorithmes. On dit cette structure est statique. Le nombre d'éléments qu'elle contient ne peut pas varier.
Dans notre exemple, nous créerons donc un tableau appelé `Note[]`. Et chaque note individuelle sera désignée par : `Note[i]` (l'élément qui se trouve à la position `i`).
Pour déclarer un tableau il faut préciser le nombre et le type de valeurs qu'il contiendra.

```
Tableau Note[25] : réels
```

Cette déclaration réserve l'emplacement de 25 éléments de type `réels`. Chaque élément est repéré par son indice (position de l'élément dans le tableau). Dans la plus part des langages de programmation (en particulier en langage C), la première position porte le numéro `0`. Dans notre exemple, les indices vont de `0` à `24`. Le premier élément du tableau sera désigné par `Note[0]`, le deuxième par `Note[1]`, le dernier par `Note[24]`. L'utilisation de `Note[25]` déclanchera une erreur.

On peut créer des tableaux contenant des variables de tous types : tableaux de numériques, tableaux de caractères, tableaux de booléens, tableaux de tout ce qui existe dans un langage donné comme type de variables. Par contre, on ne peut pas faire un mixage de types différents de valeurs au sein d'un même tableau.

L'énorme avantage des tableaux, c'est qu'on va pouvoir les traiter en faisant des boucles. Par exemple, pour effectuer notre calcul de moyenne, cela donnera par exemple :

4.1.2 Exemple

Voici un programme qui comporte la déclaration d'un tableau de 25 réels (les notes d'une classe), on commence par effectuer la saisie des notes, et en suite on calcul la moyenne des 25 notes et on affiche la moyenne :

```
variables tableau Note[25],i,somme    : entier
                      moyenne    : réel
      début
            /* saisir les notes */
         pour i allant de 0 à 24 faire
            ecrire("entrer une note :")
            lire(Note[i])
         finpour
            /* effectuer la moyenne des notes */
         somme ← 0
         pour i allant de 0 à 24 faire
            somme ← somme + Note[i]
         finPour
         moyenne = somme / 25
            /* affichage de la moyenne */
         ecrire("la moyenne des notes est :",moyenne)
      fin
```

Exécution :

```
entrer une note : 12
entrer une note : 10.5
entrer une note : 14
…   …   …   …
entrer une note : 08
la moyenne des notes est :11.75
```

Une amélioration :

```
Constante Max 200
variables tableau Note[Max],i,somme,n : entier
                          moyenne : réel
      début
         écrire("entrer le nombre de notes :")
         lire(n)
            /* saisir les notes */
         ecrire("entrer les ",n," notes :")
         pour i allant de 0 à n-1 faire
            lire(Note[i]
         finpour
            /* effectuer la moyenne des notes */
         somme ← 0
         pour i allant de 0 à n-1 faire
            somme ← somme + Note[i]
         finPour
         moyenne = somme / 25
            /* affichage de la moyenne */
         ecrire("la moyenne des ",n," notes est :",moyenne)
      fin
```

Remarque :

- A la compilation la constante `Max` sera remplacée par le nombre `200`.
- Ici le nombre de note n'est pas fixé à `25`, mais il est seulement inférieur à `200`.

- Même si on donne à `n` la valeur `10`, l'ordinateur à réserver quand même la place pour `200` variables de types réels.
- La saisie des notes est aussi améliorée, puisque on rentre sur la même ligne toutes les notes. Attention : si `n=10`, il faut rentrer `10` valeurs réelles séparer par un espace, si vous rentrez moins de dix valeurs, l'ordinateur restera bloquer, et attend que vous rentrer les dix valeurs attendues. Si au contraire vous rentrez 11 valeurs au lieu de 10, l'ordinateur affectera les10 premières valeurs au tableau `Note` et gardera la 11-ième valeur pour une prochaine lecture, ce qui provoquera peut être une erreur !!

4.1.3 Les caractéristiques de l'indice d'un tableau

L'indice qui sert à parcourir les éléments d'un tableau peut être exprimé directement comme un nombre ou une variable.

La valeur d'un indice doit toujours :

- être égale au moins à 0 (dans le langage Pascal, le premier élément d'un tableau porte l'indice 1). Mais, nous avons choisi ici de commencer la numérotation des indices à zéro, comme c'est le cas en langage C. Donc attention, `Tab[2]` est le troisième élément du tableau `Tab` !
- être un nombre entier. Quel que soit le langage, l'élément `Tab[3,1416]` n'existe jamais.
- être inférieure ou égale au nombre d'éléments du tableau moins 1. Si le tableau Tab a été déclaré comme ayant 10 éléments, la présence dans une ligne, sous une forme ou sous une autre, de Tab[10] déclenchera automatiquement une erreur.

Remarques :

1. La déclaration :
   ```
   tableau Tab[10] : entier
   ```
   créera un tableau `Tab` de 10 éléments, le plus petit indice étant 0 et le plus grand indice est 9.
2. Ne pas confondre l'**indice** d'un élément d'un tableau avec le **contenu** de cet élément. La première maison de la rue n'a pas forcément un habitant, et la dixième maison n'a pas dix habitants. En notation algorithmique, il n'y a aucun rapport entre `i` et `Tab[i]`.

Exercice 21 :

1. Ecrivez un algorithme qui lit la taille `n` d'un tableau `T`, il saisi les `n` éléments du tableau `T`, il effectue la somme des `n` éléments du tableau et il affiche cette somme.

2. Ecrivez un algorithme qui lit la taille `n` de deux tableaux T1 et T2, il effectue la lecture de ces deux tableaux, ensuite il effectue la somme des tableaux T1 et T2 dans un tableau T et il affiche le tableau T.
   Exemple :
   pour `n=8` T_1`=[4,5,8,-2,5,6,0,-5]`, T_2`=[1,-5,-7,0,-1, 3,-8,9]`,
   le tableau T obtenu est : `T=[5,0,1,-2,4,9,-8,4]`,

Exercice 22 :

1. Ecrivez un algorithme qui permet à l'utilisateur de saisir les notes d'une classe, ensuite il renvoie le nombre de ces notes supérieures à la moyenne de la classe.
2. Ecrivez un algorithme qui permet à l'utilisateur de saisir un tableau de taille n et d'afficher le plus grand et le plus petit élément du tableau.

Exercice 23 :

Que produit l'algorithme suivant ?

```
Variable Tableau F[10], i  : entier
     début
          F[0]← 1
          F[1]← 1
          écrire(F[0],F[1])
          pour i allant de 2 à 10 faire
               F[i]← F[i-1]+F[i-2]
               écrire(F[i])
          finpour
     fin
```

4.1.4 Les tableaux dynamiques

Il arrive fréquemment que l'on ne connaisse pas à l'avance le nombre d'éléments que devra comporter un tableau. Bien sûr, une solution consisterait à déclarer un tableau gigantesque (10 000 éléments) pour être sûr que "ça rentre". Mais d'une part, on n'en sera jamais parfaitement sûr (si le nombre n des éléments du tableau dépasse 10 000, ça provoquera une erreur), d'autre part, en raison de l'immensité de la place mémoire réservée (la plupart du temps non utilisée), c'est un gâchis qui affectera la taille de notre algorithme, ainsi que sa rapidité.

Pour résoudre ce problème, on a la possibilité de déclarer le tableau sans préciser au départ son nombre d'éléments. Ce n'est que dans un second temps, au cours du programme, que l'on va fixer ce nombre via une instruction d'allocation : **`Allocation(nom,nombre,type)`**, Dans la quelle il faut préciser le **nom** du tableau, le **nombre** d'éléments et le **type** des éléments à allouer. Notez que **tant qu'on n'a pas précisé le nombre d'éléments d'un tableau, d'une manière ou d'une autre, ce tableau est inutilisable**. Il ne faut pas oublier de libérer le tableau à la fin de son utilisation avec l'instruction : **`libère`**`(nom)` **.**

Exemple : on veut faire saisir des notes pour un calcul de moyenne, mais on ne sait pas combien il y aura de notes à saisir. L'algorithme sera :

```
variables tableau Note[], moyenne, somme   : réels
                                 i, n      : entiers
début
     écrire("entrer le nombre de notes à saisir : ")
     lire(n)
               /*  allouer n nombres de types réels  */
     allocation(Notes,n,réels)
               /*  saisir les notes  */
     écrire("entrer les ",n," notes :")
     pour i allant de 0 à n-1 faire
```

```
                lire(Note[i])
        finpour
                /*  effectuer la moyenne des notes  */
        somme ← 0
        pour i allant de 0 à n-1 faire
            somme ← somme + Note[i]
        finPour
        moyenne = somme / n
                /*  affichage de la moyenne  */
        écrire("la moyenne des ",n," notes est :",moyenne)
                /*  libérer  le tableau Notes  */
        libère(Notes)
    fin
```

Exercice 24 :
Refaire l'Exercice 1 précédent en utilisant des tableaux dynamiques.

4.2 Tableaux à deux dimensions

4.2.1 Introduction

Pour représenter par exemple les matrices dans un ordinateur, un tableau ne suffit pas, puisque chaque ligne de la matrice est en effet un tableau, donc une matrice n×k peut par exemple être représenter par n tableaux de k éléments chacun. Mais cette représentation sera difficile à gérer, surtout si on veut implémenter un algorithme qui effectue la multiplication ou l'addition de deux matrices. L'informatique nous offre la possibilité de déclarer des tableaux dans lesquels les valeurs ne sont pas repérées par un indice seule, mais par deux indices. C'est la solution à notre problème de représentation de matrice.
Un tel tableau se déclare ainsi :

```
tableau  matrice[10][10] : entier
```

Cette déclaration signifie : réserver un espace de mémoire pour 10 x 10 entiers, et quand j'aurai besoin de l'une de ces valeurs, je les repèrerai par deux indices.
`matrice[i][j]` est l'élément de la matrice qui se trouve à l'intersection de la ligne `i` et la colonne `j`.

```
matrice[2][3]← 5
```

Cette instruction signifie : mettre à l'emplacement qui se trouve à l'intersection de la deuxième ligne avec la troisième colonne la valeur 5.
Dans la mémoire l'ordinateur représente un tableau matrice[10][10] par un seul tableau avec la taille 10×10.

4.2.2 Initialisation de matrice :

Pour initialiser une matrice on peut utiliser par exemple les instructions suivantes :

```
T1[3][3] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
T2[3][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
T3[4][4] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
T4[4][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Ces instructions initialisent quatre tableaux de la manière suivantes :

T_1			T_2			T_3				T_4			
1	2	3	1	2	3	1	2	3	0	1	2	3	4
4	5	6	4	5	6	4	5	6	0	5	6	7	8
7	8	9	7	8	9	7	8	9	0	9	0	0	0
						0	0	0	0	0	0	0	0

Remarque : Rappelons que ces représentations rectangulaires sont très conventionnelles. Dans la mémoire de l'ordinateur ces quatre tableaux sont plutôt arrangés en lignes :

T_1 → 1 2 3 4 5 6 7 8 9
T_2 → 1 2 3 4 5 6 7 8 9
T_3 → 1 2 3 0 4 5 6 0 7 8 9 0 0 0 0 0
T_4 → 1 2 3 4 5 6 7 8 9 0 0 0 0 0 0 0

4.2.3 Lecture et écriture d'une matrice

Lecture d'une matrice :

```
     variable Tableau matrice[10][10],i,j,n,k  : entiers
     début
          écrire("donner le nombre de ligne de la matrice :")
          lire(n)
          écrire("donner le nombre de colonne de la matrice :")
          lire(k)
          pour i allant de 1 à n faire
               écrire("donner les éléments de la ",i," ligne:")
               pour j allant de 1 à k faire
                    lire(matrice[i][j])
               finpour
          fipour
     fin
```

Ecriture d'une matrice :

```
     variables Tableau Mat[10][10],i,j,n,k : entier
          début
               pour i allant de 1 à n faire
                    pour j allant de 1 à k faire
                         écrire(Mat[i][j],"  ")
                    finpour
                    écrire("\n")    /* retour à la ligne */
               finpour
          fin
```

4.2.4 Exemples d'utilisation d'une matrice

Somme de deux matrices :

```
constante N 20
variables Tableau A[N][N],B[N][N],C[N][N],i,j,n : entier
     début
          écrire("donner la taille des matrices(<20) :")
```

```
        lire(n)
                        /* lecture de la matrice A */
        pour i allant de 1 à n faire
             écrire("donner les éléments de la ",i," ligne:")
             pour j allant de 1 à n faire
                  lire(A[i][j])
             finpour
        fipour
                        /* lecture de la matrice B */
        pour i allant de 1 à n faire
             écrire("donner les éléments de la ",i," ligne:")
             pour j allant de 1 à n faire
                  lire(B[i][j])
             finpour
        fipour
                        /* la somme de C = A + B */
        pour i allant de 1 à n faire
             pour j allant de 1 à n faire
                  C[i][j] ← A[i][j]+B[i][j]
             finpour
        fipour
                        /* affichage de la matrice de C */
        pour i allant de 1 à n faire
             pour j allant de 1 à n faire
                  écrire(C[i][j],"  ")
             finpour
             écrire("\n")    /* retour à la ligne */
        finpour
   fin
```

Produit de deux matrices :

```
constante N 20
variables Tableau A[N][N],B[N][N],C[N][N],i,j,k,n,S : entier
     début
          écrire("donner la taille des matrices(<20) :")
          lire(n)
                        /* lecture de la matrice A */
          pour i allant de 1 à n faire
               écrire("donner les éléments de la ",i," ligne:")
               pour j allant de 1 à n faire
                    lire(A[i][j])
               finpour
          fipour
                        /* lecture de la matrice B */
          pour i allant de 1 à n faire
               écrire("donner les éléments de la ",i," ligne:")
               pour j allant de 1 à n faire
                    lire(B[i][j])
               finpour
          fipour
                        /* le produit de C = A * B */
```

```
        pour i allant de 1 à n faire
            pour j allant de 1 à n faire
                S ← 0
                pour k allant de 1 à n faire
                    S ← S + A[i][k]*B[k][j]
                finpour
                C[i][j] ← S
            finpour
        fipour
                    /* affichage de la matrice de C */
        pour i allant de 1 à n faire
            pour j allant de 1 à n faire
                écrire(C[i][j],"  ")
            finpour
            écrire("\n")    /* retour à la ligne */
        finpour
    fin
```

Exercice 25 :

1. Ecrire un algorithme qui effectue la lecture d'une matrice carrée `A` ainsi que sa taille `n` et affiche la trace de `A` (pour une matrice $A(a_{i,j})$, $Trace(A)=\sum a_{i,i}$ la somme des éléments sur la diagonale).
2. Ecrire un algorithme qui effectue la lecture d'une matrice carrée A ainsi que sa taille `n` et affiche la matrice transposé `tA` de `A` (Pour une matrice $A(a_{i,j})$, $tA(a_{j,i})$).

Exercice 26 :

1. Écrivez un algorithme qui effectue la lecture de :
    - `n` un entier.
    - `vect[]` un tableau de `n` nombre réels,
    - `mat[][]` une matrice carrée de `n×n` nombre réels,

    il calcule et affiche le produit de la matrice `mat` par le vecteur `vect`.
2. Écrivez un algorithme qui effectue la lecture de deux matrices allouées dynamiquement et affiche le produit de ces deux matrices.

4.3 Applications

4.3.1 Les algorithmes de recherche dans un tableau

Problème : recherche d'un élément dans un tableau.

Recherche séquentielle

Analyse :

Parcours séquentiel du tableau
Arrêt lorsque la valeur est trouvée
    Retour l'indice correspondant
La valeur n'est pas trouvée
    Retour d'une valeur spéciale

Conception :

Un programme qui demande la taille du tableau T, les éléments du tableau T, la valeur de x (la valeur cherchée), le programme donne la première position de x dans le tableau T, si x est dans le tableau T et donne la valeur -1 si x n'est pas dans le tableau T.

```
variables n,i : entier
          tableau T : réel
          x : réel
début
     écrire("donner la taille de T :")
     lire(n)
     écrire("donner les ",n," éléments de T :")
     pour i allant de 0 jusqu'à n-1 faire
          lire(T[i])
     finpour
     écrire("donner la valeur de x :")
     lire(x)
     i ← 0
     Tant que ( i < n  et  T[i] != x) faire
          i ← i + 1
     fintantque
     si (i<n) alors retourne(i)
     sinon retourne(-1)
fin
```

Test :

- Si on prend n=10, T= 1 -9 2 0 3 5 2 4 11 -1 et x=7
  la fonction retourne -1 (puisque 7 ne se trouve pas dans T !)

- Si on prend n=10, T= 1 -9 2 0 3 5 2 4 11 -1 et x=2
  la fonction retourne 2 (puisque T[2]=2 , la première position trouvée)

Recherche dichotomique

On se place dans le cas où le tableau est **ordonné** et les éléments du tableau sont deux à deux distincts.

Analyse :
x : élément recherché,
T : tableau ordonné, sans duplication,
n : la taille du tableau (le nombre d'éléments du tableau)
mil : indice du milieu du tableau,

```
si x = T[mil] alors retourne mil
sinon
     si x < T[mil] alors
          recherche dans la 1ère moitié du tableau (T[d … mil-1])
     sinon recherche dans la 2ème moitié du tableau (T[mil+1 … f])
     finsi
```

```
     finsi
     écrire(" l'élément x n'est pas dans T ")
```

Conception :

```
variables n, i, d, f, mil : entier
        tableau T : réel
              x : réel
début
     écrire("donner la taille de T :")
     lire(n)
     écrire("donner les ",n," éléments de T :")
     pour i allant de 0 jusqu'à n-1 faire
          lire(T[i])
     finpour
     écrire("donner la valeur de x :")
     lire(x)

     début
          d ← 0
          f ← n - 1
          Tantque (d<=f)
               mil ← (d + f)/2
               si (x = T[mil])
                  écrire(" l'élément x est à la position ",mil)
               sinon si (x<T[mil])  f ← mil-1
                     sinon  d ← mil + 1
                     finsi
               finsi
          fintantque
          écrire(" l'élément x n'est pas dans T ")
     fin
```

Test :

Soit le tableau T[1..9] (ordonné et sans répétition) suivant :

0	1	2	3	4	5	6	7	8
2	6	8	11	17	18	22	45	102

On cherche l'élément **8**.

d	0	0	1
f	8	3	3
mil	4	1	2

Au début : `d=0`, `f=8` comme `d<f` on rentre dans la boucle "`Tantque`", `mil=(0+8)/2=4` et comme `x<T[mil]` (8 <17) on a `f=mil-1 (f=3)`, `d=0`, puisque `d<f` (`0<3`) on reste dans la boucle `Tantque` et on a `mil =(0+3)/2=1` et comme `x>T[mil]` (8>6), donc `d=mil+1=2` et `f=3` par conséquent `d<f (2<3)`, alors on reste dans la boucle `Tantque` et `mil=2`, la `x=T[mil]` (`8=T[2]`) alors le programme affiche la valeur `2` qui est bien la position que occupe la valeur `x=8`.

Exercice 32 : Soit le tableau T[0..10] (ordonné et sans répétition) suivant :

-2	3	5	7	10	17	19	23	50	62	70

1. Chercher l'élément 3 dans le tableau T, par dichotomie.
2. Chercher l'élément 19 dans le tableau T, par dichotomie.
3. Chercher l'élément 56 dans le tableau T, par dichotomie.

5. Exercices et Problèmes d'examens

5.1 Examen d'informatique (I_2) session de Juin 2004

Enoncés des exercices :

Exercice 1 :

Ecrivez un algorithme qui demande à l'utilisateur d'entrer la température de l'eau, et affiche ensuite l'état de l'eau selon la température (on rappelle que l'état de l'eau est glace pour une température inférieure ou égale à 0°, est vapeur pour une température supérieure ou égale à 100° et liquide pour une température comprise strictement entre 0° et 100°).

Exercice 2 :

1. Ecrivez un algorithme qui lit un entier n et compte le nombre de 1 dans la représentation binaire de cet entier.
2. Écrivez un algorithme qui lit un tableau d'entiers de n éléments et donne la plus grande et la plus petite valeur de ce tableau.

Exercice 3 :

Soit la fonction mystere écrite en Maple suivante :

```
mystere := proc(n)
      local liste,x,i ;
      x := n ;
      liste := NULL ;
      i :=2 ;
      while (i<= x) do
            if (irem(x,i)=0) then
                  x := x/i;
                  liste := liste,i;
            else
                  i := i+1;
            fi;
      od;
      [liste];
end;
```

1. Que valent `mystere(8)`, `mystere(90)` et `mystere(210)` (justifier par un tableau de variables) ?
2. Que fait ce programme ?

3. Que se passera-t-il si on remplace l'instruction `i:=2;` par `i:=1;` ?

N.B : la fonction Maple `irem(x,y)` retourne le reste de la division de `x` par `y` et `x/y` désigne la division entière de `x` par `y`.

Exercice 4 :

Deux nombres entiers n et m sont qualifiés d'**amis**, si la somme des diviseurs de n est égale à m et la somme des diviseurs de m est égale à n (on ne compte pas comme diviseur le nombre lui même et 1).
Exemple : les nombres 48 et 75 sont deux nombres **amis** puisque :

- Les diviseurs de 48 sont : 2, 3, 4, 6, 8, 12, 16, 24 et
  2 + 3 + 4 + 6 + 8 + 12 + 16 + 24 = 75
- Les diviseurs de 75 sont : 3, 5, 15, 25 et 3 + 5 + 15 + 25 = 48.

Ecrire un algorithme qui permet de déterminer si deux entiers n et m sont amis ou non.

5.2 Correction de l'examen d'informatique (I_2) session Juin 2004

Exercice 1 : sur 4 points

```
Variable Temp : Entier
Début
     Ecrire("Entrez la température de l'eau :" )
     Lire(Temp)
     Si Temp =< 0  Alors Ecrire("C'est de la glace" )
     Sinon  Si Temp < 100 Alors Ecrire("C'est du liquide")
            Sinon Ecrire("C'est de la vapeur")
            finsi
      finsi
Fin
```

2ème solution (mauvaise):

```
Variable Temp : Entier
Début
     Ecrire("Entrez la température de l'eau : ")
     Lire(Temp)
     Si Temp =< 0 Alors Ecrire("C'est de la glace") finsi
     Si Temp > 0 Et Temp < 100 Alors
          Ecrire("C'est du liquide") finsi
     Si Temp > 100 Alors Ecrire("C'est de la vapeur") finsi
Fin
```

Exercice 2 : sur 7 points

1.
```
Variables i,n,poids : entiers
     Debut
          Ecrire(" Entrer la valeur de n :")
          lire(n)
          i ← n
          poids ←  0
```

```
            TantQue(i<>0) faire
                si (i mod 2 = 1) alors
                    poids ← poids + 1
                finsi
                i ← i/2
            FinTantQue
            Ecrire(poids)
        Fin
```

2.
```
variables Tableau Tab[100], i, n, min, max : Entier
debut
    ecrire("donner la taille du tableau :")
    lire(n)
    ecrire("donner les éléments du tableaux :")
    Pour i allant de 1 à n faire
        lire(Tab(i))
    FinPour
    min ← Tab[1]
    max ← Tab[1]
    Pour i allant de 2 à n faire
        si (min > Tab[i]) alors
            min ← Tab[i] finsi
        si (max < Tab[i]) alors
            max ← Tab[i] finsi
    FinPour
    Ecrire("le minimum est ",min)
    Ecrire("le maximum est ",max)
fin
```

Exercice 3 : sur 6.5 points

1. `mystere(8) :`

x	8	4	2	1
i	2	2	2	2
liste		2	2,2	2,2,2

`mystere(90) :`

x	90	45	45	15	5	5	5	1
i	2	2	3	3	3	4	5	5
liste		2	2	2,3	2,3,3	2,3,3	2,3,3	2,3,3,5

`mystere(210) :`

x	210	105	105	35	35	35	7	7	7	1
i	2	2	3	3	4	5	5	6	7	7
liste		2	2	2,3	2,3	2,3	2,3,5	2,3,5	2,3,5	2,3,5,7

2. Ce programme donne la liste des diviseurs (premiers) d’un entier n.

3. En remplaçant l'instruction `i:=2` par l'instruction `i:=1` le teste `irem(x,i)=0` est toujours vrai, `x` prend toujours la même valeur et la liste contiendra une infinité de `1` (on obtient **une boucle infini**) !

Exercice 4 : sur 3 points

1ère Solution :

```
variables n, m, d, p, S1, S2 : Entier
debut
        ecrire(" entrer l'entier n :")
        lire(n)
        ecrire(" entrer l'entier m :")
        lire(m)
        S1 ← 0
        S2 ← 0
        d ← 2
        TantQue d*d<=n faire
              Si irem(n,d)=0 alors
                    S1 ← S1 + d + n/d
              finsi
              d ← d + 1
        FinTantQue
        p ← 2
        TantQue p*p<=m faire
              Si irem(m,p)=0 alors
                    S2 ← S2 + p + m/p
              finsi
              p ← p + 1
        FinTantQue
        Si (S1=S2) alors
              Ecrire(" n et m sont amis")
        Sinon
              Ecrire(" n et m ne sont pas amis")
        finsi
fin
```

2ème Solution :

```
variables n, m, i, S1, S2 : Entier
debut
        ecrire(" entrer l'entier n :")
        lire(n)
        ecrire(" entrer l'entier m :")
        lire(m)
        S1 ← 0
        S2 ← 0
        Pour i allant de 2 à n-1 faire
              Si irem(n,i)=0 alors
                    S1 ← S1 + i
              finsi
        FinPour
        Pour i allant de 2 à m-1 faire
```

```
                Si irem(m,i)=0 alors
                     S2 ← S2 + i
                finsi
           FinPour
           Si (S1=S2) alors
                Ecrire("n et m sont amis")
           Sinon
                Ecrire("n et m ne sont pas amis")
           finsi
   fin
```

5.3 Examen d'informatique (I_2) session de Juillet 2004

Exercice 1 : **(4 points)**
Les étudiants ayant passé l'examen du module I2 en session de Juin ont été classés selon leurs notes en trois catégories :

- pour une note inférieure strictement à 5, l'étudiant est éliminé,
- pour une note supérieure ou égale à 5 et inférieur strictement à 10, l'étudiant passe la session de rattrapage,
- pour une note supérieure ou égale à 10, l'étudiant valide le module

Ecrivez un algorithme qui demande à l'utilisateur d'entrer la note du module, puis affiche la situation de l'étudiant selon sa note (on suppose que l'utilisateur entre une note valide entre 0 et 20).

Exercice 2 : **(4 points)**
Un nombre **parfait** est un entier positif supérieur à 1, égal à la somme de ses diviseurs ; on ne compte pas comme diviseur le nombre lui-même.
Exemple : 6 est un nombre parfait puisque : 6 = 3 + 2 + 1.

1. Donner un nombre parfait différent de 6.
2. Ecrire la conception de l'algorithme qui nous dit si un entier n est parfait ou non.

Exercice 3 : **(5 points)**
Soit la fonction mystere écrite en Maple suivante :

```
mystere := proc(a,b)
     local r ;
     while ( b > 0 ) do
          r := irem(a,b);
          a := b;
          b := r;
     od;
     a;
end;
```

1. Que valent `mystere(35,12)`, `mystere(96,81)` et `mystere(34,21)` (donner les valeurs que prennent les variables a, b et r dans chacun des cas) ?
2. Que fait ce programme ?

N.B : la fonction Maple `irem(x,y)` retourne le reste de la division de `x` par `y`.

Exercice 4 : **(7 points)**

1. Écrivez un algorithme qui lit la taille d'un tableau n, le tableau T, une valeur x, et il indique ensuite si l'élément x appartient ou non au tableau T.

2. Écrivez un algorithme qui permet de déterminer si les éléments d'un tableau d'entiers sont tous consécutifs ou non. (Par exemple, si le tableau est : 7; 8; 9; 10, ses éléments sont tous consécutifs. Si le tableau est : 7; 9 ; 10; 11, ses éléments ne sont pas tous consécutifs).

5.4 Correction de l'examen d'informatique (I_2) session juillet 2004

Exercice 1 :

1ère version :

```
Variables note : réel
Début
   Ecrire("Entrez la note du module :")
   Lire (note)
   Si (note < 5)   alors Ecrire ("l'étudiant est éliminé")
   Sinon Si note <10 alors
                 Ecrire ("l'étudiant passe en rattrapage")
         Sinon
                 Ecrire ("l'étudiant a validé le module")
         Finsi
  Finsi
Fin
```

2ème version :

```
Variables note : réel
Début
   Ecrire("Entrez la note du module")
   Lire (note)
   Si (note < 5)   alors  Ecrire("l'étudiant est éliminé") Finsi
   Si (note >= 5 et note < 10) alors
        Ecrire(" l'étudiant passe en rattrapage")
   Finsi
   Si (note > 10) alors Ecrire("l'étudiant a validé le module")
   Finsi
Fin
```

Exercice 2 :

1. Le nombre 28 est parfait puisque 28 = 14+4+7+2+1.
2. **solution** :

```
Variables n, d, S : entier
Début
     Ecrire ("Entrez la valeur de n : ")
     Lire (n)
     S ← 1
```

```
        d ← 2
        TantQue (d*d <= n)
              Si (n%d=0) alors S ← S + d + n/d finsi
              d ← d + 1
        FinTantQue
        Si (S=n) alors
              Ecrire ("le nombre n est parfait ")
        Sinon
              Ecrire ("le nombre n n'est pas parfait ")
        finsi
  Fin
```

Exercice 3 :

1. `mystere(35,12)` :

a	35	12	11
b	12	11	1
r	11	1	0

La valeur affichée est **1**

`mystere(96,81)` :

a	96	81	15	6
b	81	15	6	3
r	15	6	3	0

La valeur affichée est **3**

`mystere(34,21)` :

a	34	21	13	8	5	3	2
b	21	13	8	5	3	2	1
r	13	8	5	3	2	1	0

La valeur affichée est **1**

2. Cet algorithme donne le plus grand commun diviseur (le pgcd) de deux nombres entiers a et b.

Exercice 4 :

1.
```
variables n, i, x, T[20] : entiers
debut
      ecrire("entrer la valeur de n : ")
      lire(n)
      ecrire("entrer le tableau T : ")
      pour i allant de 1 à n faire  lire(T[i])  finpour
      ecrire("entrer la valeur de x : ")
      lire(x)
      i ← 1
      TantQue ( i ≤ n  et  T[i] ≠ x) faire
            i ← i + 1
      FinTantQue
      Si (i ≠ n+1) alors
         Ecrire(" l'élément x se trouve dans le tableau T ")
      Sinon
         Ecrire("l'élément x ne se trouve pas dans le tableau T ")
      finsi
fin
```

2. Dans cette question, on suppose que les variables `n` et `T` sont connues.

1ère version:

```
Variable i: entier
Début
  i ← 1
  TantQue (i<n ET T[i+1] = T[i]+1)
     i ← i+1
  FinTantQue
  Si (i<n) alors
     Ecrire("les termes du tableau ne sont pas consécutifs")
  Sinon
     Ecrire(" les termes du tableau sont consécutifs ")
  FinSi
Fin
```

2ème version:

```
Variable i, j: entier
Début
   j ← 1
   Pour i allant de 1 à n-1
        Si (T[i+1] ≠ T[i] +1) alors
               j ← 0
                i ← n-1
        FinSi
   FinPour
   Si (j=1) alors
     Ecrire("les termes du tableau sont consécutifs")
  Sinon
    Ecrire("les termes du tableau ne sont pas consécutifs")
  Finsi
Fin
```

5.5 Examen d'informatique (I_2) session de Juin 2005

Exercice 1 :

Ecrire un algorithme qui demande l'âge d'un enfant à l'utilisateur. Ensuite, il l'informe de sa catégorie :

- La catégorie d'un enfant est "Poussin" si $6 \leq age < 8$,
- La catégorie d'un enfant est "Pupille" si $8 \leq age < 10$,
- La catégorie d'un enfant est "Minime" si $10 \leq age < 12$,
- La catégorie d'un enfant est "Cadet" si $age \geq 12$.

Exercice 2 :

1. Ecrivez un algorithme qui lit un entier n et compte le nombre de 1 dans la représentation binaire de cet entier.
2. Écrivez un algorithme qui lit un entier n (la taille du tableau), le tableau d'entiers T de n éléments, l'entier x et indique le nombre de fois que x figure dans le tableau T.

3. Ecrivez une procédure qui prend pour arguments un entier n (la taille du tableau), le tableau de réels T et affiche le plus grand élément du tableau T ainsi que sa position dans le tableau. (On suppose que le tableau T est formé d'éléments tous distincts)

Exercice 3 :
Soit la fonction mystere écrite en Maple suivante :

```
>   mystere := proc(L,t)
      local u, i;
      u := 0;
      for i from 1 to nops(L) do
      u := u*t + L[i];
      od;
      u;
end;
```

1. Que valent `mystere([1,-1,0,2],3)`, `mystere([1,0,1,1,1],2)` et `mystere([a,b,c],x)` (donner la valeur de `u` à chaque étapes de la boucle) ?
2. Que fait ce programme ?
3. Que calcule la procédure `mystere(L,t)` si les éléments de la liste `L` sont tous strictement inférieur à `t` ?

Exercice 4 :
Soit T un tableau qui contient n valeurs réelles triés dans l'ordre croissant. Ecrire une procédure qui prend comme paramètre le Tableau T, l'entier n (la taille de T) et un nombre réel x, et elle effectue l'insertion de x dans le tableau T, de telle manière que le tableau T reste trié.

Exemple : Soit le tableau T de 8 nombres triés dans lequel on désire insérer le nombre 40 :

4	7	8	12	23	56	89	112

Le résultat est un tableau T de 9 nombres toujours triés :

4	7	8	12	23	40	56	89	112

5.6 Examen d'informatique (I_2) session de Juin 2006

Exercice 1 : **3 points**
Dans le championnat de foot marocain (qui comporte 16 équipes) les deux premières équipes sont qualifiées pour la ligue des champions africaine, les deux suivants (le troisième et le quatrième) sont qualifiés pour la ligue des champions arabes, les trois dernières équipes descendent en deuxième division, et les autres restent en première division.
Ecrire un algorithme qui demande le classement d'une équipe du championnat de foot marocain. Ensuite, il nous informe de sa situation pour l'année prochaine.

Correction :

```
Nom : Foot
Role : donne la situation d'une équipe en fonction de son classement
Entrée : n un entier compris entre 1 et 16
Sortie : un message
Variables : n entier
Debut
     écrire("donner votre classement (entre 1 et 16) :")
     lire(n)
```

```
    si n<=0 alors
         écrire(" faux classement !")
    else si n<=2 alors
        écrire(" l'équipe ira à la ligue des champions africaine ")
           else si n<=4 alors
               écrire(" l'équipe ira à la ligue des champions arabe ")
           else si n<=14 alors
                    écrire(" l'équipe restera en première division ")
                else
                    écrire(" l'équipe jouera en deuxième division ")
                finsi
           finsi
        finsi
    finsi
fin
```

Exercice 2 : 10 points

1.
   a. Écrivez une fonction `fact()`, qui prend pour argument un entier `k` et retourne le factoriel de `k`.
   b. Écrivez une procédure qui effectue la lecture d'un entier n, calcule et affiche la somme des factoriels des entiers de `0` à `n` : $\sum_{i=0}^{n} i!$. (utiliser la fonction `fact()` précédente).
2. Ecrivez une fonction qui prend en argument un entier `n` et retourne le nombre de `1` dans la représentation binaire de l'entier `n`.
3. Écrivez une procédure qui lit un entier n (la taille du tableau), le tableau d'entiers T de n éléments, l'entier x et indique le nombre de fois que x figure dans le tableau T.

Correction :

1. a. **2 points**

```
fonction fact(n entier) : entier
var f,i :entier
debut
     f←1
     i←1
     tantque(i<=n) faire
          f←f*i
          i←i+1
     fintantque
     retourne(f)
fin
```

b. **2 points** nom : somme factorielle

rôle : calcule de $\sum_{i=0}^{n} i!$

```
entrée : n :entier
sortie : S : entier
variable : n, S, i : entiers
debut
     écrire("donner n :")
     lire(n)
```

```
                S←0
                i← 1
                tanque(i<=n) faire
                     S←S+fact(i)
           i←i+1
                fintanque
                écrire(" la somme est ",S)
           fin
```

2. sur **3 points** :
```
           fonction  poids(n :entier) :entier
           variables i,n,poids : entiers
           début
                i ← n
                poids ←  0
                TantQue(i<>0) faire
                     si (i mod 2 = 1) alors
                          poids ← poids + 1
                     i ← i/2
                FinTantQue
                Ecrire(poids)
           Fin
```

3. sur **3 points** :
```
variables Tableau Tab[100], i, n, x, c : entier
debut
     ecrire("donner la taille du tableau :")
     lire(n)
     ecrire("donner les éléments du tableaux :")
     Pour i allant de 1 à n faire
          lire(Tab(i))
     FinPour
     ecrire("donner x :")
     lire(x)
     c ← 0
     Pour i allant de 0 à n-1 faire
          si (x = Tab[i]) alors c ← c+1 finsi
     FinPour
     Ecrire("le mombre de",x," est ",c)
fin
```

Exercice 3 : 4 points

Soit la fonction `mystere()` écrite en Maple suivante :
```
>   mystere := proc(n)
     local s, i;
     s := 0, 1;
     i:=3;
     while(i<=n) do
          s := s, s[i-1]+s[i-2];
          i:=i+1;
     od;
     [s];
end;
```

1. Que valent `mystere(10)` ?

2. Que fait ce programme ?
3. On remarque que pour calculer s[i] on besoin que de s[i-1] et s[i-2], comment améliorer cette fonction (écrire une fonction qui n'utilise pas de liste `s`) ?

Correction :

1. sur **1 point** : `mystere(10) : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34`
2. sur **0.5 point** : cette procédure calcule les termes de la suite de Fibonacci.
3.

```
fonction  Fibo(n :entier)
variables i, n, u, v, w : entiers
début
      si (n=1) alors écrire(0," ")
      si (n=2) alors écrire(1," ")
      i ← 3
      TantQue(i<=n) faire
            écrire(u," ")
            w ← v
            v ← u + v
            u ← w
            i ← i+1
      FinTantQue
Fin
```

Exercice 4 : **3 points**

On désire écrire un algorithme qui affiche l'image miroir `n'` d'un nombre entier `n`.
Exemple : l'image miroir du nombre `n=54321` est `n'=12345`.

1. Analyse : expliquer comment extraire à partir de l'entier `n` les chiffres qui le compose.
2. Ecrire une conception de l'algorithme qui lit un entier `n` et affiche son image miroir.

Correction :

1. Pour extraire les chiffres qui composent un entier n il suffit de répéter les deux instructions suivantes :

```
r ← n mod 10      n ← n/10
```

2.

```
nom : miroir
rôles : affichage de l'image miroir d'un entier
entrée : n :entier
sortie : les chiffres qui composent n
variables n, i, r : entier
début
      écrire("Entrer un entier : ")
      lire(n)
      tanque(n<>0) faire
            r ← n mod 10
            écrire(r)
            n ← n/10
      fintantque
fin
```

Exercice 1 :

1. **1.** Écrivez un algorithme qui effectue la lecture de trois nombres et affiche le plus grands de ces trois nombres.
2. **2.** Écrivez un algorithme qui effectue la lecture de n nombres, ensuite affiche le plus grands de ces n nombres (n saisie au clavier).

Correction :

1. 1ère solution (chercher une solution avec le minimum de tests) :

```
Nom :        maximum
Rôle :       donne le maximum de trois nombres
Entrée :     a, b, c : nombres
Sortie :     message
Variables a, b, c : nombres
Début
       ecrire("donner trois nombres :")
       lire(a,b,c)
       si (a>=b) alors
              si (a>=c) alors
                     ecrire("le maximum est ",a)
              sinon
                     ecrire("le maximum est ",c)
              finsi
       sinon si (b>=c) alors
                     ecrire("le maximum est ",b)
               sinon
                     ecrire("le maximum est ",c)
       finsi
fin
```

2ème solution :

```
Nom :        maximum
Rôle :       donne le maximum de trois nombres
Entrée :     a, b, c : nombres
Sortie :     max : nombre
Variables a, b, c, max : nombres
Début
       ecrire("donner trois nombres :")
       lire(a,b,c)
       max ← a
       si (b>max) alors max ← b finsi
       si (c>max) alors max ← c finsi
       ecrire("le maximum est ",max)
fin
```

2.

```
Nom :        maximum
Rôle :       donne le maximum de trois nombres
Entrée :     a: nombre
Sortie :     max : nombre
Variables a, max : nombres
Début
       ecrire("donner l'entier n :")
       lire(n)
       ecrire("donner le premier nombre :")
       lire(max)
       pour i allant de 2 à n faire
              ecrire("donner le ",i," nombre :")
              lire(a)
              si (a>max) alors max ← a finsi
       finpour
       ecrire("le maximum est ",max)
fin
```

Exercice 2 : Soit l'algorithme suivant :

```
variables a,b,r : entiers
début
      écrire("donner les valeurs de a et b : ")
      lire(a,b)
      TantQue (b>0) faire
            r ← a%b       /* a%b :reste de la division de a par b */
            a ← b
            b ← r
      FinTanQue
      écrire(a)
Fin
```

1. Exécuter le programme précédent (afficher dans un tableau les valeurs de a, b et r) pour :
   a. a = 35 et b = 12
   b. a = 96 et b = 81
   c. a = 34 et b = 21
2. Implémenter ce programme en langage C ;
3. Que fait ce programme.

Correction :

1. `mystere(35,12) :`

a	35	12	11	**1**
b	12	11	1	0
r	11	1	0	

La valeur affichée est **1**

`mystere(96,81) :`

a	96	81	15	6	**3**
b	81	15	6	3	0
r	15	6	3	0	

La valeur affichée est **3**

`mystere(34,21) :`

a	34	21	13	8	5	3	2	**1**
b	21	13	8	5	3	2	1	0
r	13	8	5	3	2	1	0	

La valeur affichée est **1**

2.
```
#include<stdio.h>
#include<conio.h>
void main(){
      int a,b,r ;
      printf("donner a et b :") ;
      scanf("%d %d",&a,&b) ;
      while(b !=0){
            r = a%b       /* a%b :reste de la division de a par b */
            a = b
            b = r
      }
      printf("le pgcd est %d\n",a) ;
      getch() ;
}
```

Exercice 3 :

Deux nombres entiers n et m sont qualifiés d'**amis**, si la somme des diviseurs de n est égale à m et la somme des diviseurs de m est égale à n (on ne compte pas comme diviseur le nombre lui même et 1).

Exemple : les nombres 48 et 75 sont deux nombres **amis** puisque :

Les diviseurs de 48 sont : 2, 3, 4, 6, 8, 12, 16, 24 et

$$2 + 3 + 4 + 6 + 8 + 12 + 16 + 24 = 75$$

Les diviseurs de 75 sont : 3, 5, 15, 25 et

$$3 + 5 + 15 + 25 = 48.$$

Ecrire un programme qui permet de déterminer si deux entiers n et m sont amis ou non.

Correction :

```
 variables n, m, d, p, S1, S2 : Entier
debut
        ecrire(" entrer les entiers n et m :")
        lire(n,m)
        S1 ← 0
        S2 ← 0
        d ← 2
        TantQue (d*d<=n) faire
              Si (n%d=0) alors  S1 ← S1 + d + n/d   finsi
              d ← d + 1
        FinTantQue
        p ← 2
        TantQue (p*p<=m) faire
              Si (m%p)=0 alors  S2 ← S2 + p + m/p  finsi
              p ← p + 1
        FinTantQue
        Si (S1=S2) alors
              Ecrire(" n et m sont amis")
        Sinon
              Ecrire(" n et m ne sont pas amis")
fin
```

Exercice 4 (un nombre parfait) :

Un nombre **parfait** est un entier positif supérieur à 1, égal à la somme de ses diviseurs ; on ne compte pas comme diviseur le nombre lui-même.

Exemple : 6 est un nombre parfait puisque : 6 = 3 + 2 + 1.

1. Donner un nombre parfait différent de 6.
2. Donner l'analyse de l'algorithme correspondant à la fonction parfait().
3. Donner la conception de cet algorithme.

Correction :

1. 28 est un nombre parfait puisque : 28 = 14 + 7 + 4 + 2 + 1.
2. Pour un entier n donné, on calcul ses diviseurs ainsi que la somme de ses diviseurs, si la somme est égale à n alors le nombre n est parfait, sinon il ne l'est pas.
3.
```
s et d : entiers
ecrire("donner n :")
lire(n)
```

```
s ← 1
d ← 2
tant que (d*d < n) faire
   si (n%d=0) alors s ← s + d + n/d finsi
   d ← d + 1
finTantque
si (s=n) alors ecrire("n est parfait")
sinon ecrire("n n'est pas parfait")
finsi
```

4.6 Examen d'Informatique (juin 2011)

Exercice 1 : (6.5 points)

I. QCM (2.5 points) **:** Cocher **la bonne réponse** :

1. Le nombre décimal **4321_{10}** est codé en base 16 par :		
	1141_{16}	A
	$10E1_{16}$	B
	$1E01_{16}$	C
2. le nombre de bits nécessaire pour coder en binaire le nombre décimal 3000 est :		
	13 bits	A
	12 bits	B
	11 bits	C
3. L'entendu du codage sur 6 bits en binaire signé est compris entre :		
	0 et **63**	A
	-32 et **31**	B
	-31 et **31**	C
4. Quelle est la valeur décimale du nombre **$10010110_{cà1}$** écrits en complément à 1 **:**		
	-105	A
	-104	B
	-103	C
5. Le plus grand nombre entier qu'on peut coder (en complément à 2) sur 10 bits est :		
	1023	A
	512	B
	511	C

II. Codage (4 points)

1. Coder les nombres entiers relatifs du tableau suivant sur **1 octet** :

Nombre	Binaire signé	Complément à 2
$62_{(10)}$		
$-76_{(10)}$		

2. Évaluer le nombre réel en format IEEE 754 simple précision : **1 10000100 00011100000000000000000**

Exercice 2 : **(4 points)** Soit le fragment de programme suivant :

```
variables u,v : entiers
début
     écrire("donner les valeurs de u et v : ")
     lire(u,v)
     TantQue (u != v) faire
          Si (u > v) alors u ← u - v  Sinon v ← v - u finsi
     FinTanQue
     écrire(u)
Fin
```

1. Donner les valeurs de u et v après chaque itération dans le cas suivant :

u	24					
v	14					

2. Donner les valeurs de u et v après chaque itération dans le cas suivant :

u	96										
v	81										

3. Que fait ce programme ?

Exercice 3 : **(4.5 points)**

Soit T un tableau qui contient n valeurs réelles. Ecrire un programme qui lit l'entier n (la taille de T), le tableau T, ensuite le programme affiche si les éléments du tableau T sont tous **consécutifs** ou non.

Exemples :

1. Soit le tableau T_1 de 8 éléments :

9	10	11	12	13	14	15	16

Le programme affiche : les éléments du tableau T_1 sont consécutifs.

2. Soit le tableau T_2 de 6 nombres :

5	10	40	66	99	112

Le programme affiche : les éléments du tableau T_2 ne sont pas consécutifs.

Exercice 4 : **(5 points)**

Un nombre **parfait** est un entier positif supérieur à 1, égal à la somme de ses diviseurs ; on ne compte pas comme diviseur le nombre lui-même.
Exemple : 6 est un nombre parfait puisque : 6 = 3 + 2 + 1.

4. Donner un nombre parfait différent de 6.
5. Ecrire un programme qui effectue la lecture d'un entier n et affiche si n est parfait ou non.